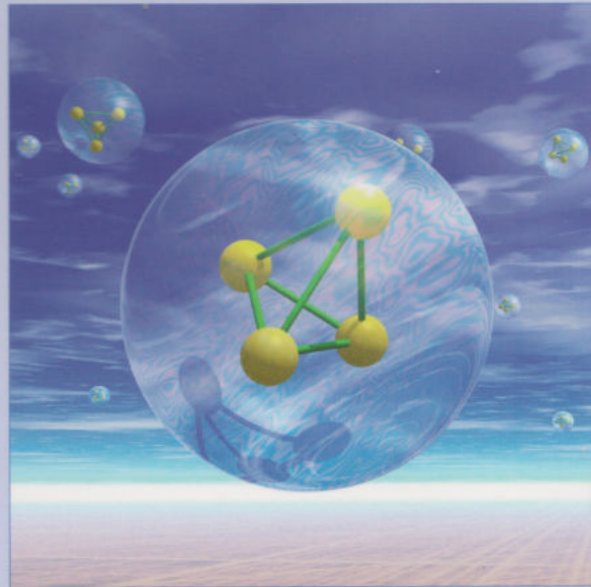




The Open University

M255 Unit 9
UNDERGRADUATE COMPUTING

Object-oriented programming with Java



Collections: Arrays,
strings and
StringBuilders

Unit **9**



M255 Unit 9

UNDERGRADUATE COMPUTING

Object-oriented programming with Java



Collections: Arrays,
strings and
StringBuilders

Unit 9

This publication forms part of an Open University course M255 *Object-oriented programming with Java*. Details of this and other Open University courses can be obtained from the Student Registration and Enquiry Service, The Open University, PO Box 197, Milton Keynes, MK7 6BJ, United Kingdom: tel. +44 (0)870 333 4340, email general-enquiries@open.ac.uk

Alternatively, you may visit the Open University website at <http://www.open.ac.uk> where you can learn more about the wide range of courses and packs offered at all levels by The Open University.

To purchase a selection of Open University course materials visit <http://www.ouw.co.uk>, or contact Open University Worldwide, Michael Young Building, Walton Hall, Milton Keynes, MK7 6AA, United Kingdom for a brochure: tel. +44 (0)1908 858785; fax +44 (0)1908 858787; email ouwenq@open.ac.uk

The Open University
Walton Hall
Milton Keynes
MK7 6AA

First published 2006. Second edition 2008. Third edition 2009.

Copyright © 2006, 2008, 2009 The Open University.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, transmitted or utilised in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without written permission from the publisher or a licence from the Copyright Licensing Agency Ltd. Details of such licences (for reprographic reproduction) may be obtained from the Copyright Licensing Agency Ltd of 90 Tottenham Court Road, London, W1T 4LP.

Open University course materials may also be made available in electronic formats for use by students of the University. All rights, including copyright and related rights and database rights, in electronic course materials and their contents are owned by or licensed to The Open University, or otherwise used by The Open University as permitted by applicable law.

In using electronic course materials and their contents you agree that your use will be solely for the purposes of following an Open University course of study or otherwise as licensed by The Open University or its assigns.

Except as permitted above you undertake not to copy, store in any medium (including electronic storage or use in a website), distribute, transmit or retransmit, broadcast, modify or show in public such electronic materials in whole or in part without the prior written consent of The Open University or in accordance with the Copyright, Designs and Patents Act 1988.

Edited and designed by The Open University

Typeset by The Open University

Printed and bound in the United Kingdom by The Charlesworth Group, Wakefield

ISBN 978 1 8487 3137 0

3.1

The paper used in this publication contains pulp sourced from forests independently certified to the Forest Stewardship Council (FSC) principles and criteria. Chain of custody certification allows the pulp from these forests to be tracked to the end use (see www.fsc.org).



CONTENTS

Introduction	5
1 The array	7
1.1 The structure of an array	7
1.2 Declaring array variables and creating array objects	9
1.3 Putting elements into an array	12
1.4 Accessing the components of an array	19
2 Processing arrays	25
2.1 Iterating through an array	25
2.2 Sub-array processing	31
2.3 Inserting an element into a sorted array	33
2.4 Two-dimensional arrays	38
2.5 <code>java.util.Arrays</code>	45
3 The <code>main()</code> method	49
3.1 Developing programs outside the BlueJ environment	49
3.2 Arguments to the <code>main()</code> method	51
3.3 A complete program that uses arrays	53
3.4 Programs that involve more than one class	56
4 Strings	57
4.1 Creating and manipulating <code>String</code> objects	57
4.2 Equality and identity	59
4.3 <code>String</code> objects are immutable	62
5 The <code>StringBuilder</code> class	63
5.1 Creating <code>StringBuilder</code> objects	63
5.2 The protocol of <code>StringBuilder</code>	65
5.3 Revisiting the concatenation operator, <code>+</code>	66
6 Summary	68
Glossary	70
Index	72

M255 COURSE TEAM

Affiliated to The Open University unless otherwise stated.

Rob Griffiths, Course Chair, Author and Academic Editor

Lindsey Court, Author

Marion Edwards, Author and Software Developer

Philip Gray, External Assessor, University of Glasgow

Simon Holland, Author

Mike Innes, Course Manager

Robin Laney, Author

Sarah Mattingly, Critical Reader

Percy Mett, Academic Editor

Barbara Segal, Author

Rita Tingle, Author

Richard Walker, Author and Critical Reader

Robin Walker, Critical Reader

Julia White, Course Manager

Ian Blackham, Editor

Phillip Howe, Composer

John O'Dwyer, Media Project Manager

Andy Seddon, Media Project Manager

Andrew Whitehead, Graphic Artist

Thanks are due to the Desktop Publishing Unit, Faculty of Mathematics and Computing.

Introduction

So far you have mainly been concerned with individual objects or individual primitive values. In many real-life applications we have collections of related data, either objects or primitive values, which we want to group together. For example:

A meteorological office may want to store monthly rainfalls for a particular location.

A hospital ward may want to keep a list of all of its patients.

We are familiar with different ways of structuring data in real life. For example we often use lists or tables to structure large amounts of data. There are structures that have rules about the ordering and accessing of the data. For example, a printer queue, where new jobs are added to the end of the queue, and the first job in the queue is the next to be sent to the printer. Or a tree structure, such as the kind used to store folders and files on a computer.

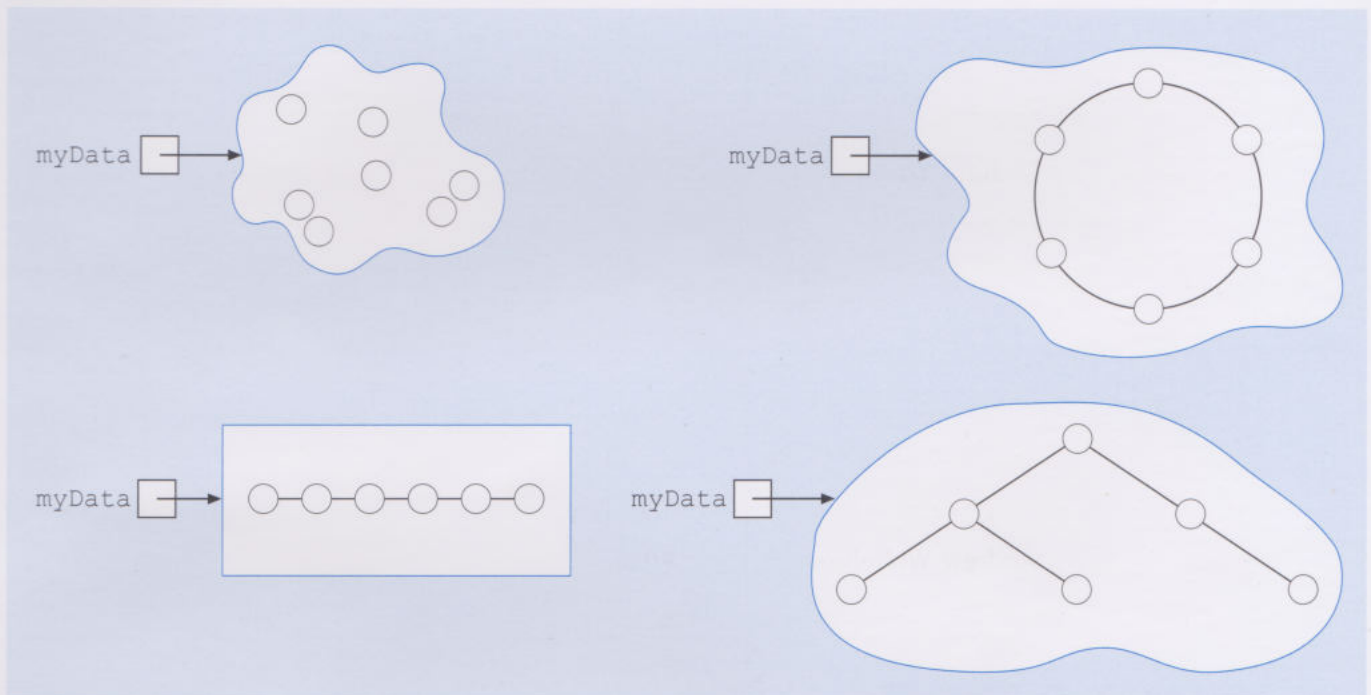


Figure 1 Different ways of structuring data

Java provides many different ways of structuring data. In this unit we will look at two kinds of object: **array** and **string** objects, which are common to most programming languages.

An array can be imagined as a series of numbered pigeonholes, each of which contains a value or a reference to an object, where the number of pigeonholes is fixed in advance.

A string is a sequence of characters that cannot be changed once the string has been created.

We will also look at the `StringBuilder` class, whose instances are like strings, but ones in which characters can be changed and added.

In addition, in *Unit 10* and *Unit 11* you will learn about a whole family of classes called the **Collection classes**, which provide many other ways of structuring and accessing data.

Some activities in this unit are accompanied by a text file. Such files may be opened within the OUWorkspace (so you will not need to type in large amounts of code). You may also wish to save your answers in this file. Where appropriate, a text file has also been included with the answers to the activity (so you can run the correct code without typing it in).

1

The array

An array is an object, and we will refer to arrays as objects in this unit to reinforce this, but in fact there is no single array class, instead, the class of an array object is automatically generated at compile time as we shall show later. As you might guess from this, arrays are a fundamental, built-in part of Java, and as such are designed to be a very efficient way of storing and accessing data. Because array data structures were developed early on in the history of programming languages, a well-established syntax for dealing with them had long been developed, well before Java was on the scene. Because a large community was familiar with this traditional syntax, Java has adopted it, even though it is not consistent with the rest of the language syntax. In most of this section we will use the traditional array syntax, but in Section 2.5 we will visit the class `java.util.Arrays`, which contains various methods for manipulating arrays using a more Java-like syntax.

1.1 The structure of an array

The distinctive features of an **array** are:

- ▶ it is a **fixed-size** structure;
- ▶ it can store either objects or primitive values (unlike the collection classes you will see in later units, which can store only objects);
- ▶ it is a **homogeneous** structure, which means that each reference or primitive value held by an array is of the same type;
- ▶ it is an **indexable** structure; every item in an array is accessed by an integer **index**.

Here is an example of an array that is being used to store `double` primitive values:

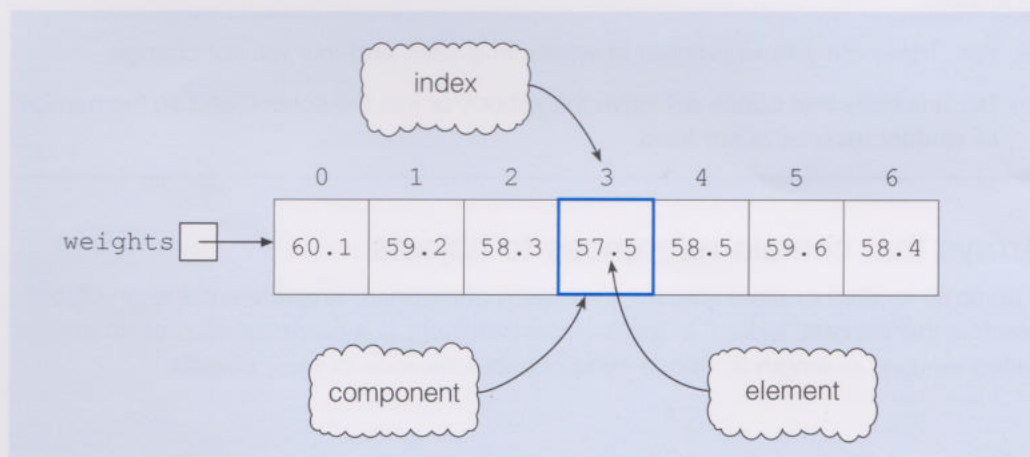


Figure 2 The parts of an array

The array is represented as a row of boxes labelled 0, 1, 2, 3, 4, 5, 6. The correct term for one of these integer labels is 'index', and a box is referred to as a **component**. You can think of each component as being similar to a variable – it is simply a memory location that can hold either a primitive value or a reference to an object. Unlike the variables you have come across so far there is not an identifier associated with the memory location, rather the programmer can identify a component using the name of the array and the index of the component (we will see how this is done in Section 1.3). Each component

Note that the index is not part of the item being stored in the component, but rather it is a means of identifying a component.

In Java the first component of an array always has an index of 0. This is called zero-based indexing.

holds an **element**; in this example the elements are values of type `double`. The variable `weights` is referencing the whole array object.

When we want to identify an element of an array we say '60.1 is the element held by the component labelled by index 0' or in shorthand '60.1 is at index 0'.

SAQ 1

In Figure 2,

- (a) What is the index of the component holding the element 57.5?
- (b) What element is held by the component at index 5?

ANSWER.....

- (a) The index of the component holding the element 57.5 is 3.
 - (b) 59.6 is at index 5.
-

We mentioned that an array is always of fixed size. By fixed size we mean that the number of components in an array must be specified when the array object is created. Thereafter the array will always have that number of components, never more, never less. This means that you should only use an array when you know that the number of elements you want it to hold will not change after it has been created; you will come across flexible size collections in *Unit 10* that may be more appropriate if this is not the case.

SAQ 2

- (a) A business wants to store its revenue for a year, broken down in weekly sales figures. Is an array a suitable **collection** structure for this task?
- (b) A school wants to store records for each of its pupils. Is an array structure a suitable collection in which to store the records?

ANSWER.....

- (a) Yes. There are a fixed number of weeks in a year, and this will not change.
 - (b) No. It is likely that pupils will leave the school, or join the school, and so the number of student records is not fixed.
-

Arrays that contain references to objects

If an array is used to store objects, then each component is a reference to an object which is the element 'stored' in that component. Here is a representation of an array called `roster`, of length 3, whose components reference `Strong` objects:

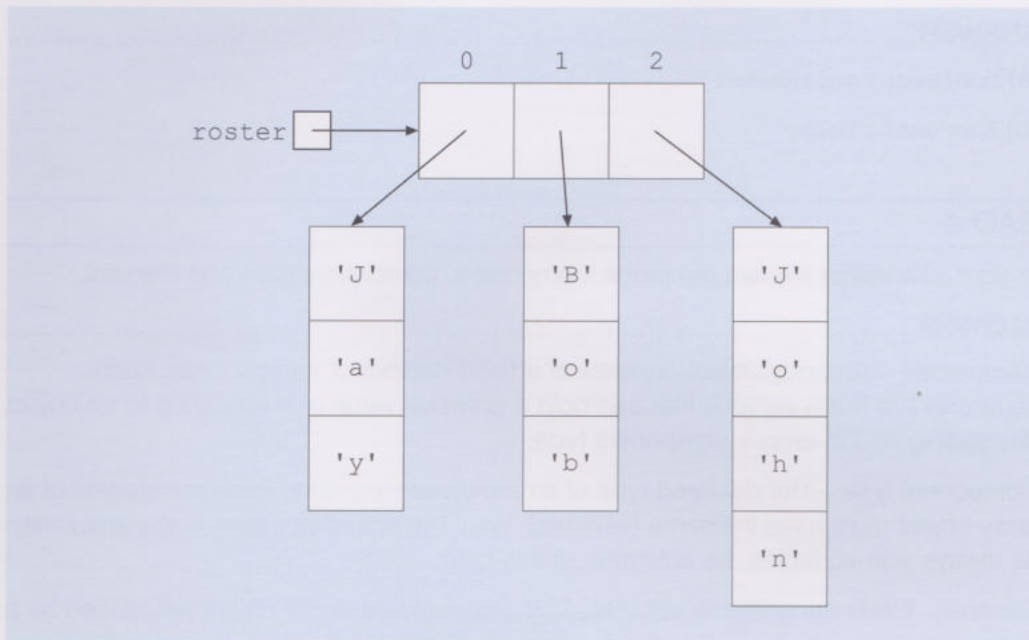


Figure 3 An array containing references to `String` objects

A reference to the string "Jay" is held in the component at index 0; a reference to the string "Bob" is held in the component indexed by 1, etc.

1.2 Declaring array variables and creating array objects

Declaring an array variable

Suppose we want to declare an array variable, `roster`, which can reference an array of `String` objects. Here is the statement that we would use:

```
String[] roster;
```

Like the declarations you have previously seen, this one comes in two parts. There is a type, `String[]`, and the name of the variable, `roster`. However, notice the use of the square brackets, `[]`, as a part of the type `String[]`. It is these square brackets that indicate that this statement declares an array variable. Whenever we declare an array variable we have to include details of the type of data that will be stored in the array (in this case `String`). This is referred to as the **component type** of the array. The type of every element stored in an array must be the same as the component type of the array or a subtype of the component type.

The identifier `roster`, gives the name of the array variable. The variable `roster` can be used to reference any array object that contains references to `String` objects. However a declaration such as this does not itself create an array object; one way of creating a new array object is to use the `new` operator which you have used for creating other types of objects.

The square brackets are a distinctive feature of array syntax.

SAQ 3

- Write a declaration for an array variable, `quizAnswer`, which can reference an array object whose elements are `boolean` values.
- Write a declaration for an array variable, `bank`, which can reference an array object whose elements are `Account` objects.

ANSWER.....

(a) `boolean[] quizAnswer;`

(b) `Account[] bank;`

SAQ 4

In your own words explain the terms: component, component type and element.

ANSWER.....

Component – An array object consists of a fixed number of components. Each component is like a variable that can hold a primitive value or a reference to an object depending on the array's component type.

Component type – The declared type of an array's components. Each component of an array object must have the same (variable) type. The actual elements of the array may, as always with variables, be subtypes of this type.

Element – Either the primitive value held by a component or the object referenced by a component.

Creating an array

Once an array variable has been declared, a new array object can be created and assigned to it. Here is the code that will associate the array variable, `roster` that we previously declared, with a newly created array, with seven components that can hold references to seven `String` objects (elements):

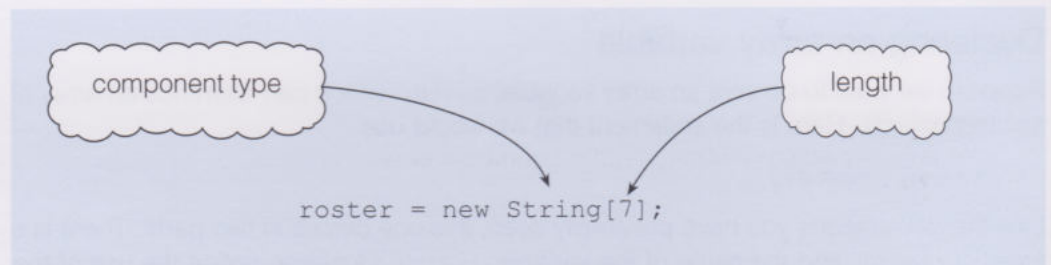


Figure 4 Creating an array and assigning it to a variable

This statement creates an array *object* that has 7 components each of which can reference any `String` object, and causes the previously declared array variable, `roster`, to reference this new array. When a newly created array is assigned to a variable, its component type must be the same as the component type that has been indicated in the declaration of that variable (or a subtype of that variable's component type).

Because of zero-based indexing the length of an array is always one more than the last index.

Notice that in order to create the array object we need to indicate the fixed number of elements it can store. This integer, which appears inside the square brackets, is referred to as the **length** of the array. On creation of a new array object, its length is automatically stored in a public instance variable which can be accessed using the usual dot notation: `roster.length`. Note that `length` in this expression is not a message, it is an attribute (an instance variable) of the array object, so it is `length` and not `length()`. When creating an array object, the length (the number of components we want) can be any expression that evaluates to an `int`. For example, these are valid statements:

```

int noOfElements = 7;
roster = new String[noOfElements];
  
```


or

```
int num = 6;
roster = new String[num + 1];
```

It is important to realise that the creation of the array referenced by `roster` does not actually create seven strings or even allocate memory to hold seven strings (as they are at present of unknown size). However, enough space to store *references* to seven strings has been allocated in memory. Although the components of the newly created array do not yet contain references to any strings they are automatically initialised on creation. If the component type of the array is an object, each component is initialised to `null`; if the component type is one of the numerical primitive types (`double`, or `int`, for example), then each component is initialised to `0.0` or `0` as appropriate. If the component type is a `boolean`, the components are initialised to `false`, and if it is a `char`, the components are initialised to the null character. These are the same default values that are used to initialise instance variables that are not otherwise initialised in a constructor.

When an array object is created, the array components allocated in memory consist of contiguous memory locations – in other words each of the memory locations after the first comes immediately after the previous one. It is this property of arrays that makes them efficient, as it makes it easy to determine quickly the memory location identified by a particular index.

Combined declaration and creation

Often the declaration of an array variable, and the assignment of a newly created array object to it, are combined into a single statement:

```
String[] roster = new String[7];
```

SAQ 5

- (a) Write a single statement that both declares a variable `bankBalances`, and assigns to it an array object that can hold 52 numbers of type `double`.
- (b) Write a single statement that both declares a variable `frogPond`, and assigns to it an array object that can hold references to three `Frog` objects.

ANSWER.....

- (a) `double[] bankBalances = new double[52];`
- (b) `Frog[] frogPond = new Frog[3];`

A shortcut for creating and initialising arrays

There is a useful shortcut for creating a new array that assigns values to each component using literal array syntax. Here is an example:

```
String[] roster = {"Jay", "Bob", "John", "Anne", "Ali", "Agi", "Jill"};
```

On the left of the assignment statement an array variable `roster` is declared. The expression on the right creates an array of length 7, and populates it with the `String` objects referenced by the string literals separated by commas within the braces.

Here is another example where each component is initialised with a reference to a different newly initialised `Frog` object:

```
Frog[] frogPond = {new Frog(), new Frog(), new Frog()};
```

The null character is a character with a Unicode or ASCII value of 0. It is essentially a character that does nothing at all. If sent to a printer it is often output as `□`.

Note that this syntax can only be used if the literal array is assigned to an array variable in the same statement as the array variable is declared. Thus the following will not compile:

```
Frog[] frogPond;
frogPond = {new Frog(), new Frog(), new Frog()};
```

as the second line is not valid.

However the following variation is valid.

```
Frog[] frogPond;
frogPond = new Frog[]{new Frog(), new Frog(), new Frog()};
```

allowing us to use the literal array syntax *and* to separate the declaration of the array variable from the creation and initialisation of the array object. This is important to us in the case of array *instance variables* as it allows us to treat an array instance variable in the same way as other instance variables with the declaration being separated from the initialisation which takes place in the constructor.

SAQ 6

In a single statement, declare an array variable, *hours*, with a component type of *int*, and assign to it an array that has each of its 5 components initialised to 40.

ANSWER.....

```
int[] hours = {40, 40, 40, 40, 40};
```

We stated in the Introduction to this unit that array classes are automatically generated at compile time. This is what occurs.

When the Java compiler comes across the declaration of an array such as:

```
String[] roster;
```

it generates an array class whose instances can hold elements of the declared component type – in this case *String* objects. This new class will be a direct subclass of *Object* and is given the name (for internal use) of `[Ljava.lang.String;` which is the name that the Java compiler always gives to a class whose instances are arrays that can hold strings. This can all be shown by executing the following code in the *OUWorkspace*:

```
String[] roster = new String[3];
roster.getClass().getName();
roster.getClass().getSuperclass().getName();
```

which will print in the Display Pane:

```
"[Ljava.lang.String;"
"java.lang.Object".
```

Similarly, the class of an array that holds *Frog* objects is named `[LFrog;` by the compiler. For array classes whose instances will hold values of some primitive type the compiler uses a slightly different naming convention; for arrays of *int* values, the class name is `[I` and for arrays of *double* values the class name is `[D`. You do not need to remember these implementation details or the names of these array classes, but you will see these names used in the titles of *Inspector* windows if you inspect a variable declared as some array type *before* it has been assigned an actual array object.

1.3 Putting elements into an array

Suppose the following code is executed:

```
double[] weights = new double[7];
```

to create an array of length 7 referenced by the variable `weights`. Because the component type is a numerical primitive type, specifically type `double`, the elements are initialised to `0.0`:

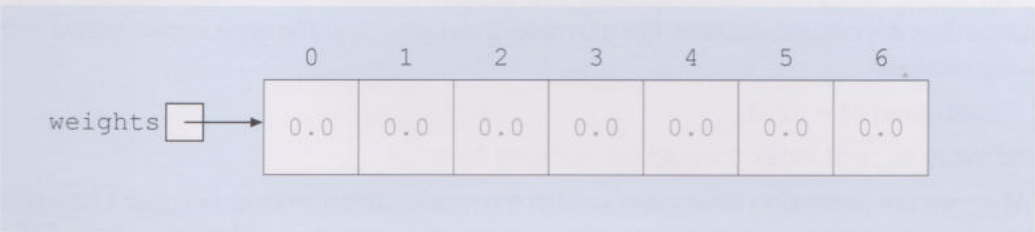


Figure 5 The newly created array, referenced by the variable `weights`

Each component of an array is really just like a variable, but instead of using an identifier to access it we use the name of the array, for example the variable `weights`, followed by the appropriate index in square brackets. For example, this expression: `weights[0]` refers to the component at index 0:

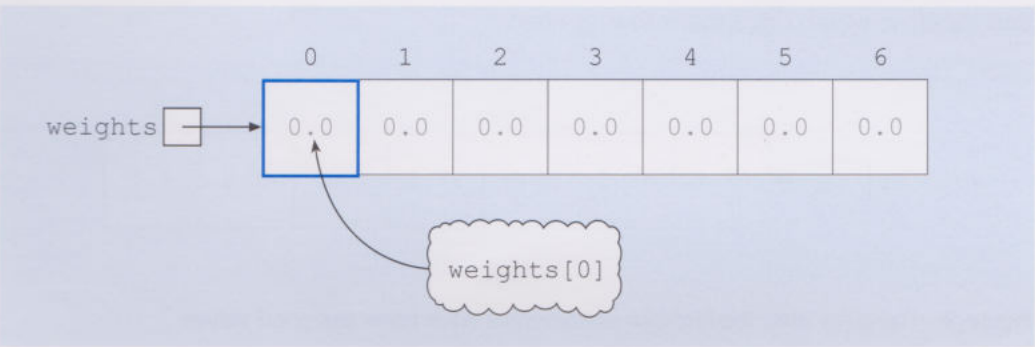


Figure 6 Referencing the first component

We can use an expression like this wherever we might use a variable. If, for example, we want to assign a value to this component we can write:

```
weights[0] = 60.1;
```

Just as with any other assignment expression, the value of the expression on the right is assigned to the component indicated on the left. In this case the `double` value `60.1`, is assigned to the component at index 0:

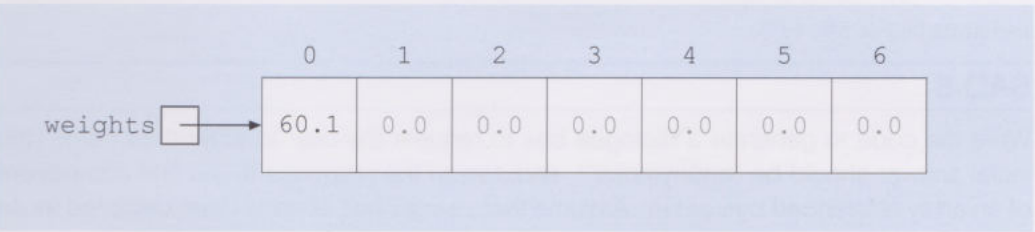


Figure 7 The array after the first component has been assigned a value

Similarly, the statement

```
weights[1] = 59.2;
```

puts the value `59.2` at index 1:



Figure 8 The array after the first two components have been assigned values

If an item is assigned to a component that already contains an element, the new item overwrites the original element. For example, if `weights` is in the state shown above and we executed:

```
weights[0] = 62.8;
```

the value 60.1 at index 0 would be replaced by 62.8.

When we are accessing array components the index can be a literal `int`, but it can also be an expression that evaluates to an `int` within the valid range of index values. For example, given:

```
int num = 2;
```

The following statements are both valid:

```
weights[num] = 58.3;  
weights[num + 1] = 57.5;
```

and together would have the following effect:

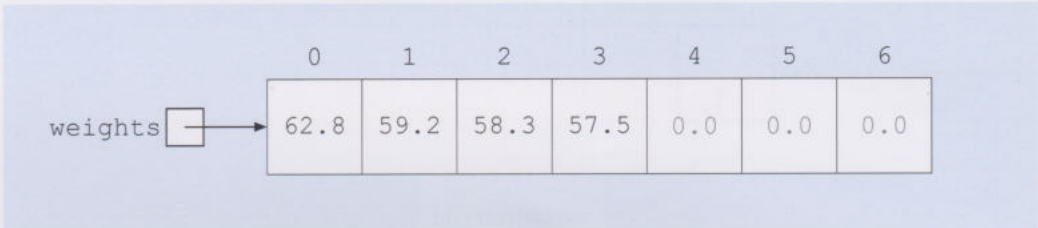


Figure 9 The array after the first four components have been assigned values

SAQ 7

Write three assignment statements to assign values of type `double` to the remaining three components of the array referenced by `weights` (as it is shown in Figure 9) so that it matches Figure 2.

ANSWER.....

```
weights[4] = 58.5;  
weights[5] = 59.6;  
weights[6] = 58.4;
```

SAQ 8

Write the code to generate a dialogue box to request the user to enter their name (the initial answer should be "anonymous"), and assign the response to the first component of an array referenced by `roster`. Assume that `roster` has already been declared as an array of strings.

ANSWER.....

```
roster[0] = OUDialog.request("Please enter your name", "anonymous");
```

Note that 'first component' always means the component with index 0.

Although it is possible to assign elements to any random component of an array, most often an array is filled sequentially, in the order in which the user inputs elements (starting with index 0, then 1, etc.). If there are fewer values than the length of the array this can lead to partially filled arrays, such as the following:

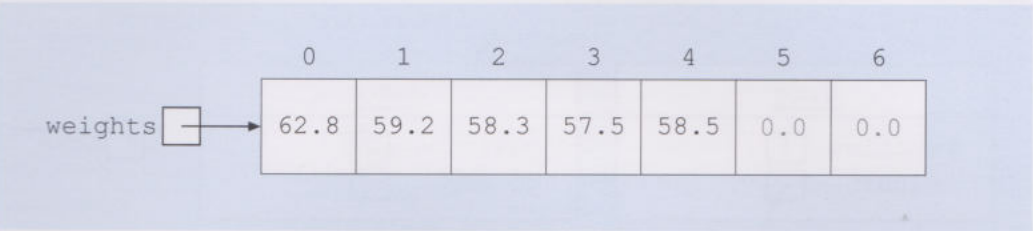


Figure 10 A partially filled array

Under these circumstances not all of the components contain real data, some still contain their initialisation values (here 0.0), so the length of the array is greater than the number of meaningful elements in the array. We often want to know how many meaningful elements there are in an array, and one way to do this is to keep a count in some variable as the elements are stored in the array. Such a count is sometimes called the **effective length** of the array. For the array in Figure 10 the effective length would be 5.

Storing values given by non-literal expressions in arrays

Suppose that we declared a variable and created an array to hold Frog objects, as follows:

```
Frog[] frogPond = new Frog[3];
```

Here is one way we could assign a newly initialised Frog object to the component of the array at index 0:

```
frogPond[0] = new Frog();
```

Alternatively we could also use a temporary variable to reference the new Frog object before assigning it to an array component. For example, if we wanted to put a reference to a frog with position 2 and colour RED into the component at index 1 we could write the following code:

```
Frog temp;  
temp = new Frog();  
temp.setPosition(2);  
temp.setColour(OUColour.RED);  
frogPond[1] = temp;
```

The component at index 1 now references a Frog object but that same Frog object is also still referenced by the variable temp as shown in Figure 11 overleaf:

Note that instead of writing 'the component at index 1 of frogPond references the Frog object' we can simply write 'the element at index 1 of frogPond is the Frog object'. The element at an index of an array object is simply the value stored in that component (if the element is a primitive value) or the object referenced by that component.

Technically the figure should have the form of Unit 3 Figure 14, where the instances of `OUColour` objects are drawn explicitly. Here, to simplify the diagrams, `OUColour` objects are represented by a box with the name of a colour in it.

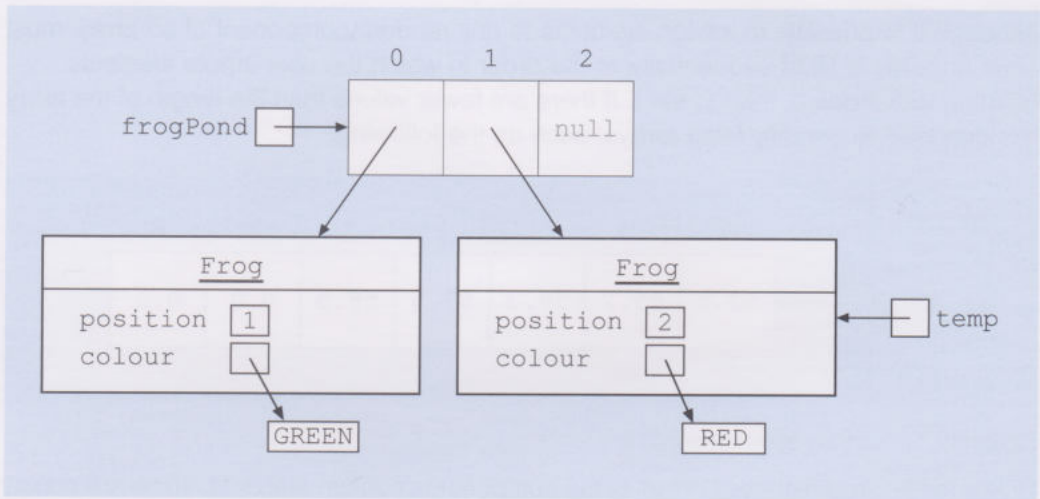


Figure 11 An array containing references to `Frog` objects

If the following statements are executed:

```
temp.right(); temp.brown();
```

the state of the `Frog` object referenced by `temp` will change, which means that the component at index 1 of the array will also reference this changed object:

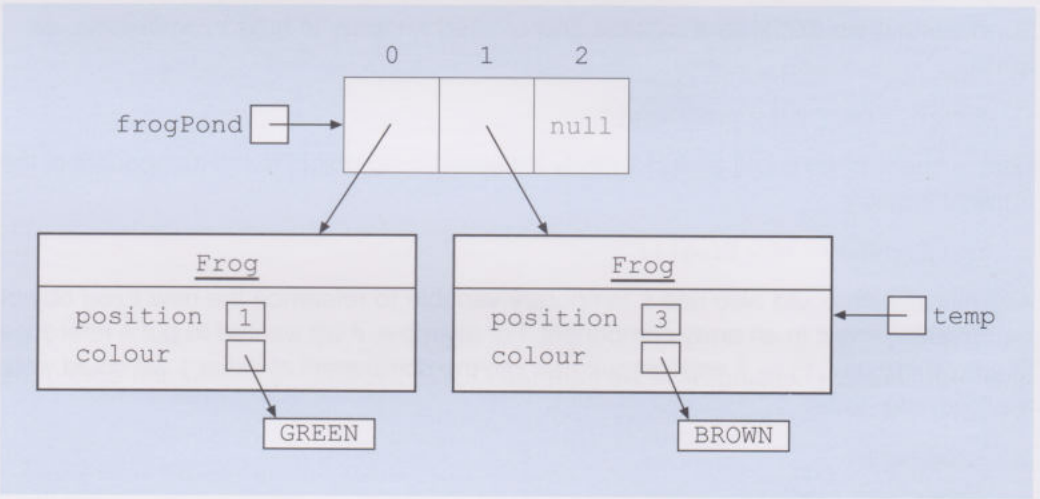


Figure 12 An array containing references to `Frog` objects

Substitutability

In Unit 6 you discovered the property of substitutability – that is, it is legal to assign an object to a variable declared to be of the type of one of its superclasses. For example, the following is a legitimate assignment to the variable `kermit`:

```
Frog kermit = new HoverFrog();
```

The same applies to array components: it is possible to assign an object to an array component that has been declared to be the type of one of that object's superclasses. This means that the following is legal:

```
Frog[] frogPond = new Frog[3];
frogPond[2] = new HoverFrog();
```

The component type of the array is of type `Frog` and so only messages that are in the protocol of `Frog` can be sent to the components of `frogPond`. However, the reference held in the array at index 2 is to an object of type `HoverFrog`, so if `HoverFrog` has overridden any of the `Frog` methods, it is the `HoverFrog` versions that are invoked.

SAQ 9

Is the following a valid declaration and initialisation of an array?

```
Object[] anArray = {new HoverFrog(), new HoverFrog()};
```

ANSWER.....

Yes, anArray has been declared with component type Object, which is a superclass of HoverFrog.

ACTIVITY 1

Launch BlueJ and open project Unit9_Project_1. Then from the Tools menu select OUWorkspace.

Make sure that the Show results checkbox is checked and then write code to do the following. Note what is returned in each case, and inspect the variables to check that your code works as expected.

- 1 Create a new array object, referenced by nameArray, that can hold references to six String objects. Then inspect nameArray.
- 2 Write a statement that will return the length of nameArray.
- 3 Assign the following strings to the components of nameArray (in the given order, starting at index 0). "Ann" "Rob" "Kin" "Sue" "Fethi" "Jo".
- 4 Replace the string element "Sue", that is stored in nameArray, with "Lin".
- 5 Replace the name at index 2 of nameArray with a name entered by the user using a dialogue box.
- 6 Create an array referenced by intArray that contains the integers 10, 5, 7, 2, 9, 8, 1 and 12 (in this order).
- 7 Create an array, frogPond that can hold references to three Frog objects, and assign a new Frog object to each component (the Frog objects should be in their initial state).

If you wish, after finishing this Activity, you could save the contents of the Code Pane for later retrieval, by choosing Save As from the OUWorkspace's file menu and saving the contents with a suitable file name.

DISCUSSION OF ACTIVITY 1

- 1 String[] nameArray = new String[6];
You should see the following when you inspect nameArray:

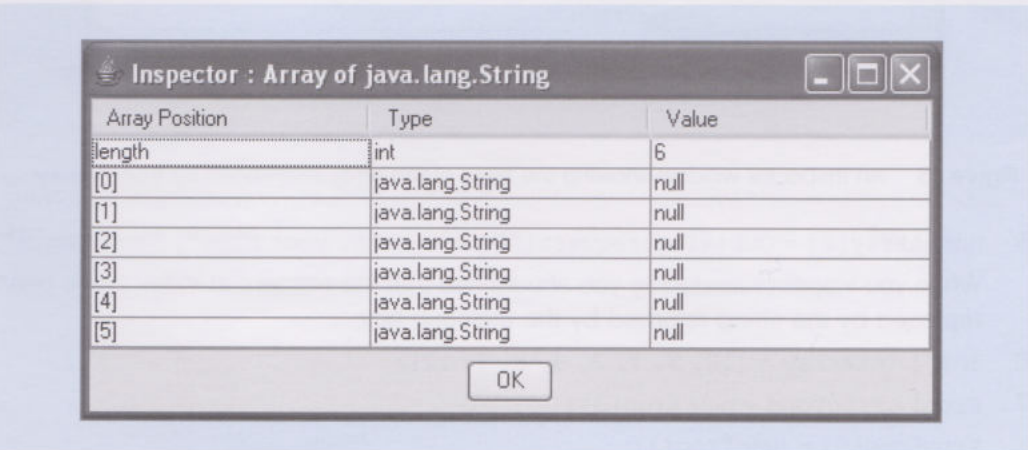


Figure 13 An Inspector window showing the state of the array referenced by nameArray

2 Executing

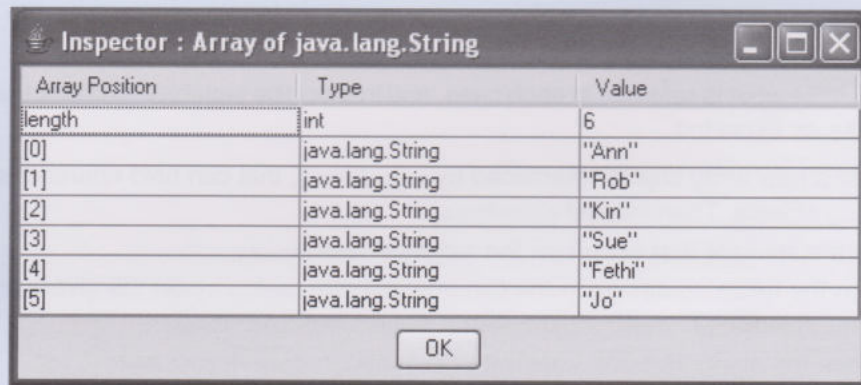
```
nameArray.length;
```

will return the length of the array.

3 You should have executed the following series of statements:

```
nameArray[0] = "Ann";
nameArray[1] = "Rob";
nameArray[2] = "Kin";
nameArray[3] = "Sue";
nameArray[4] = "Fethi";
nameArray[5] = "Jo";
```

You should see the following when you inspect nameArray.

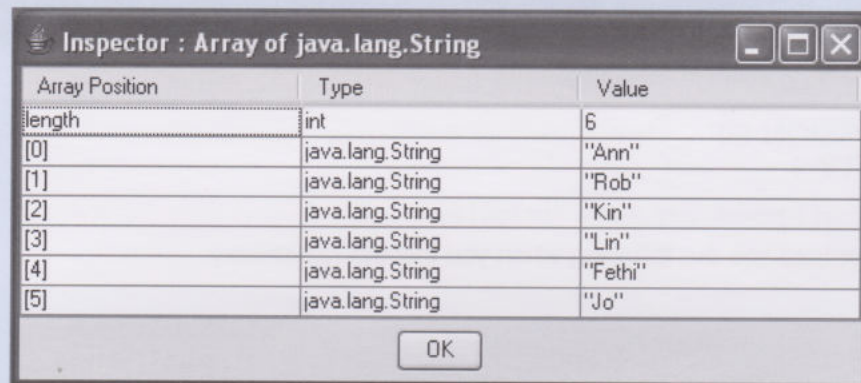


Array Position	Type	Value
length	int	6
[0]	java.lang.String	"Ann"
[1]	java.lang.String	"Rob"
[2]	java.lang.String	"Kin"
[3]	java.lang.String	"Sue"
[4]	java.lang.String	"Fethi"
[5]	java.lang.String	"Jo"

Figure 14 An Inspector window showing the state of the array referenced by nameArray

4 nameArray[3] = "Lin";

You should see the following when you inspect nameArray.



Array Position	Type	Value
length	int	6
[0]	java.lang.String	"Ann"
[1]	java.lang.String	"Rob"
[2]	java.lang.String	"Kin"
[3]	java.lang.String	"Lin"
[4]	java.lang.String	"Fethi"
[5]	java.lang.String	"Jo"

Figure 15 An Inspector window showing the state of the array referenced by nameArray

5 nameArray[2] = OUDialog.request("Please enter your name", "anonymous");

When you inspect nameArray you should find that the element at index 2 has been replaced by the string returned by the dialogue box.

6 int[] intArray = {10, 5, 7, 2, 9, 8, 1, 12};

7 Frog[] frogPond = new Frog[3];

```
frogPond[0] = new Frog();
frogPond[1] = new Frog();
frogPond[2] = new Frog();
```


Alternatively:

```
Frog[] frogPond = {new Frog(), new Frog(), new Frog()};
```

In either case when you inspect `frogPond` you should see the following.

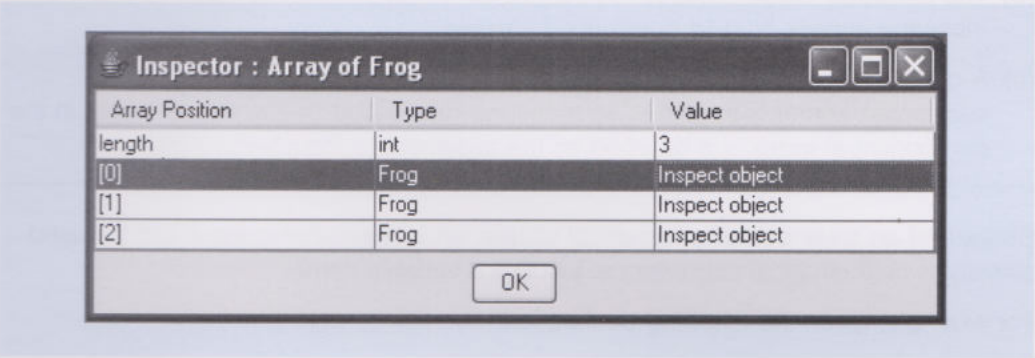


Figure 16 An Inspector window showing the state of the array referenced by `frogPond`

1.4

Accessing the components of an array

An indexed component of an array, can be used anywhere that a variable of the same type can be used (except if the element references null, in which case a `NullPointerException` may be thrown).

So, given the array, `roster`, created as follows:

```
String[] roster = {"Jay", "Bob", "John", "Anne", "Ali", "Agi", "Jill"};
```

The following are all valid:

- 1
- ```
OUDialog.alert(roster[2]);
```

Effect: displays the string at index 2 of `roster`, "John", in a dialogue box.
- 2
- ```
String weekendWorkers = roster[0] + " and " + roster[6];
```

Effect: assigns to the variable `weekendWorkers` the string formed by concatenating the string referenced by the component at index 0 of `roster` with the string " and " followed by the string at index 6 of `roster` (so `weekendWorkers` references "Jay and Jill").
- 3
- ```
roster[5].equals(roster[6]);
```

Effect: returns `true` if the string at index 5 of `roster` has the same state as the string at index 6 of `roster`, and `false` otherwise (so `false` would be returned in this case).

SAQ 10

Given the array, `roster`, created as follows:

```
String[] roster = {"Jay", "Bob", "John", "Anne", "Ali", "Agi", "Jill"};
```

Write down what would result from executing the following statements/expressions.

- (a)
- ```
System.out.println(roster[2]);
```
- (b)
- ```
String name = roster[6].toUpperCase();
```
- (c)
- ```
OUDialog.alert(roster[roster.length - 2]);
```

ANSWER.....

- (a) Outputs the string "John" to the default output device. (So, if executed in the OUWorkspace this would be the Display Pane.)
- (b) The message `toUpperCase()` is sent to the `String` object at index 6, and the message answer "JILL" is assigned to `name`.
- (c) A dialogue box displays the string "Agi". (`roster.length` returns 7, 2 is subtracted from this to give 5, so the string referenced by the element held in the component at index 5 is returned as the argument to the `alert()` method.)

Similarly, if an array contains numerical values, an indexed component can be used directly in mathematical expressions, just like a variable name.

For example, given the following declaration:

```
int[] intArray = {7, 9, 3};
```

The following are all valid.

- 1 `intArray[1] + intArray[2];`
Effect: $9 + 3$ is evaluated and the result 12 is returned.
- 2 `int num = (intArray[1] - intArray[0]) / 2;`
Effect: the right-hand side is evaluated as $(9 - 7) / 2$ so the `int` result, 1, is assigned to `num`.
- 3 `intArray[0] = intArray[0] + 1;`
Effect: the right-hand side is evaluated as $7 + 1$, and the result, 8, is assigned to the component at index 0.

The component type of the array is `int`, and so integer division is performed here.

SAQ 11

Given the following code:

```
double[] doubleArray = {2.5, 3.5, 2.0, 2.0};
```

Describe the effect of each of these expressions (assume that the array is initialised as above before each expression is evaluated):

- (a) `(doubleArray[1] + doubleArray[0]) / doubleArray[3]`
- (b) `doubleArray[3] = doubleArray[0] + doubleArray[1] + doubleArray[2]`
- (c) `(int) doubleArray[0] > (int) doubleArray[3]`

ANSWER.....

- (a) $(3.5 + 2.5) / 2$ is evaluated and the result is 3.0.
- (b) $(2.5 + 3.5 + 2.0)$ is evaluated and the result is 8.0. This is assigned to the component at index 3 of `doubleArray`.
- (c) The value of `doubleArray[0]` is cast to an `int`, which gives the result 2. The value of `doubleArray[3]` is cast to an `int`, which gives the result 2. Finally $2 > 2$ is evaluated, and the result is false.

You came across type casting in *Unit 3*.

Accessing objects in arrays

Consider the array referenced by `frogPond`, which contains references to three `Frog` objects:

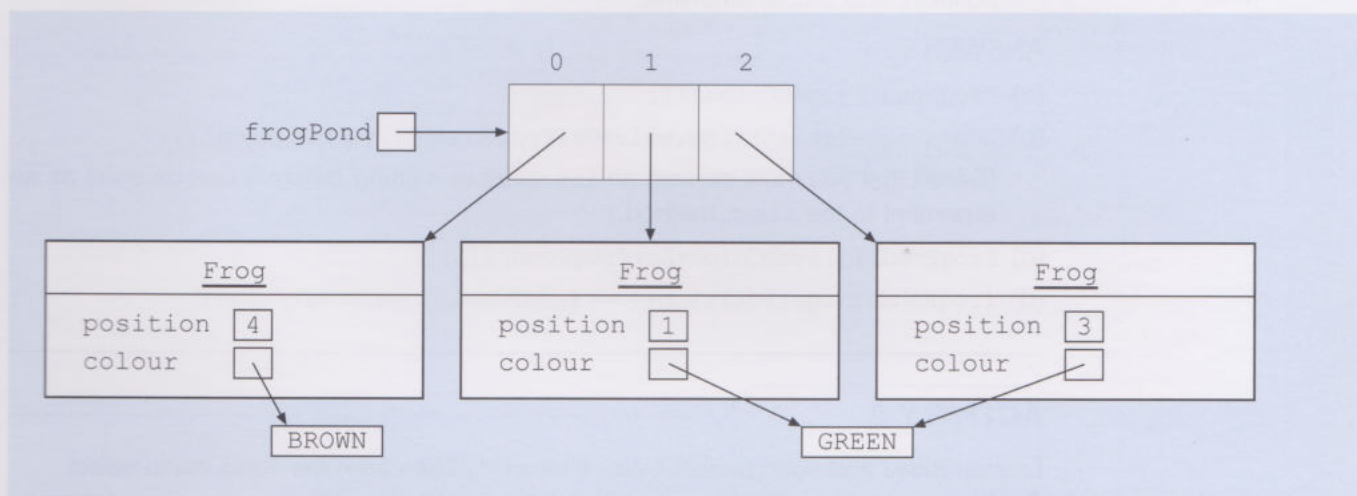


Figure 17 An array containing `Frog` objects

The expression `frogPond[1]` references the `Frog` object at index 1. We can then use the familiar dot notation to send a message to this `Frog` object. Of course, the message we send must be in the protocol of `Frog` objects.

So, to change the colour of the `Frog` object at index 1 to RED, and its position to 2, we would write the following code:

```
frogPond[1].setColour(OUColour.RED);
frogPond[1].setPosition(2);
```

The array can now be represented like this:

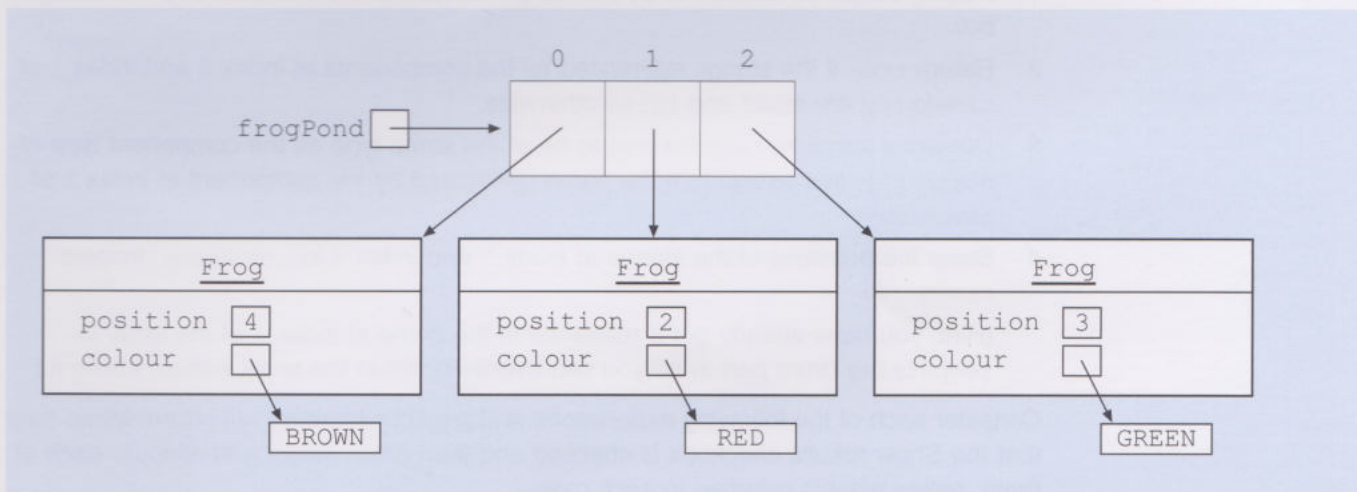


Figure 18 An array containing `Frog` objects

SAQ 12

Given an array, `frogPond`, containing references to three `Frog` objects, write code that does the following.

- Gets the colour of the `Frog` object at index 2.
- Displays the position of the `Frog` object at index 0 in a dialogue box.

- (c) Changes the colour of the `Frog` object at index 1 to be the same as the colour of the `Frog` object at index 2.
- (d) Evaluates `true` if the two `Frog` objects at index 1 and index 2 have the same position, and `false` otherwise.

ANSWER.....

- (a) `frogPond[2].getColour();`
- (b) `OUDialog.alert(String.valueOf(frogPond[0].getPosition()));`
(Recall that you must convert an `int` value to a string before it can be used as an argument to the `alert` method.)
- (c) `frogPond[1].sameColourAs(frogPond[2]);`
- (d) `frogPond[1].getPosition() == frogPond[2].getPosition();`

ACTIVITY 2

Launch BlueJ and open project `Unit9_Project_1`. Then from the Tools menu select `OUWorkspace`. Open the file called `CodeForActivity2`. This will save you some typing by loading into the Code Pane some of the code needed for this activity. Select and execute the following statements, and then inspect each variable. For `frogPond` you should also inspect each component of the array:

```
String[] nameArray = {"Ann", "Rob", "Kin", "Sue", "Fethi", "Jo"};
int[] intArray = {10, 5, 7, 2, 9, 8, 1, 12};
Frog[] frogPond = {new Frog(), new Frog(), new Frog()};
```

Make sure that the Show results checkbox is checked and then enter, select and execute statements to do each of the following, noting what is returned in each case, and inspecting the variables to ensure that your code worked as expected.

- 1 Display the name referenced by the component at index 4 of `nameArray` in a dialogue box.
- 2 Return `true` if the strings referenced by the components at index 0 and index 1 of `nameArray` are equal and `false` otherwise.
- 3 Declare a variable, `tempString`, to be of the same type as the component type of `nameArray`, and assign to it the name referenced by the component at index 1 of `nameArray`.
- 4 Swap the positions of the strings at index 1 and index 4 in `nameArray`. Inspect `nameArray`.

(Hint: you have already got a reference to the name at index 1 of the array in `tempString` (from part 3) so you can overwrite this in the array without losing it.)

Consider each of the following expressions and predict what they will return. Make sure that the Show results checkbox is checked and then enter, select and execute each of them, noting what is returned in each case.

- 5 `intArray[0] + intArray[6];`
- 6 `intArray[7] / intArray[2];`
- 7 `intArray[0] + intArray[4] * intArray[3];`
- 8 `intArray[4] / 2.0;`

Enter, select and execute statements to do each of the following and inspect the variables to ensure that your code worked as expected.

- 9 Replace the element at index 7 of `intArray` with a value twice its current value.

- 10 Replace the element at index 3 of `intArray` with the sum of the values at indexes 1 and 2.

Enter, select and execute statements to do each of the following, and inspect `frogPond` (and the relevant components of `frogPond`) to ensure that your code worked as expected.

- 11 Change the colour of the `Frog` object at index 2 of `frogPond` to red.
 12 Cause the `Frog` object at index 0 of `frogPond` to move right.
 13 Change the colour of the `Frog` object at index 0 of `frogPond` to be the same as the colour of the `Frog` object at index 2 of `frogPond`.
 14 Add 3 to the current position of the `Frog` object at index 2 of `frogPond`.
 15 Swap the `Frog` object at index 1 with the `Frog` object at index 0 of `frogPond`. (You will need to declare a temporary variable to do this.)

DISCUSSION OF ACTIVITY 2

```
1 OUDialog.alert(nameArray[4]);
```

```
2 nameArray[0].equals(nameArray[1]);
```

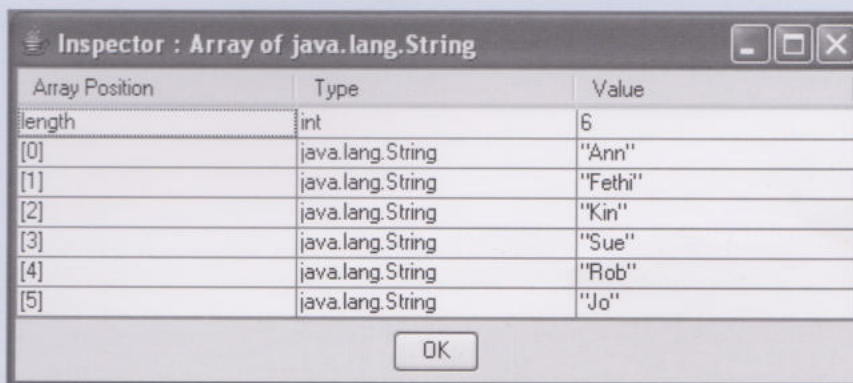
Recall that the message `equals()` (and not `==`) should be used when checking if two strings represent the same sequence of characters.

```
3 String tempString = nameArray[1];
```

```
4 nameArray[1] = nameArray[4];  
  nameArray[4] = tempString;
```

In part 3 you assigned the string at index 1 to the variable `tempString`. Here (in part 4) the first line assigns the element at index 4 to the component at index 1 (overwriting it). The second line assigns the `String` object referenced by `tempString` to the component at index 4.

You should see the following when you inspect `nameArray`.



Array Position	Type	Value
length	int	6
[0]	java.lang.String	"Ann"
[1]	java.lang.String	"Fethi"
[2]	java.lang.String	"Kin"
[3]	java.lang.String	"Sue"
[4]	java.lang.String	"Rob"
[5]	java.lang.String	"Jo"

OK

Figure 19 An Inspector window showing the state of the array referenced by `nameArray`

- 5 11 (10 + 1)
- 6 1 (12 / 7) integer division.
- 7 28 (10 + 9 * 2) multiplication is done first, then the addition.
- 8 4.5 (9 / 2.0) the operand 2.0 of type double forces this to be floating-point division.
- 9 `intArray[7] = intArray[7] * 2;`
- 10 `intArray[3] = intArray[1] + intArray[2];`
- 11 `frogPond[2].setColour(OUColour.RED);`
`frogPond[2]` references the Frog at index 2, and it is sent the message `setColour(OUColour.RED)`.
- 12 `frogPond[0].right();`
`frogPond[0]` references the Frog at index 0, and it is sent the message `right()`.
- 13 `frogPond[0].sameColourAs(frogPond[2]);`
`frogPond[0]` references the Frog at index 0, and it is sent the message `sameColourAs()` with the argument being the Frog object at index 2.
- 14 `frogPond[2].setPosition(frogPond[2].getPosition() + 3);`
The expression in parentheses (the argument of the message `setPosition()`) gets the position of the Frog object at index 2 and adds 3 to it. This value is returned and becomes the argument for the message `setPosition()` which is sent to the Frog object at index 2.
- 15 `Frog tempFrog = frogPond[1];`
`frogPond[1] = frogPond[0];`
`frogPond[0] = tempFrog;`
This code is similar to the code in part 4 but notice that the variable `tempFrog` must be declared to be of the same type as the component type of the array (which here is `Frog`).
-

2

Processing arrays

2.1 Iterating through an array

When dealing with arrays we often want to process each component in the same way. For example, we may want to initialise each component to the same value, or print out each component's element in turn. Because we always know (or can at least find out) the number of components in an array we can use a `for` loop for this kind of iteration.

For example, consider this array:

```
int[] myArray = new int[2000];
```

which we can depict as follows.



Figure 20 An array of length 2000

We can initialise each component to 10 using the following `for` loop:

```
for (int i = 0; i < myArray.length; i++)
{
    myArray[i] = 10;
}
```

In the first iteration, `i` is 0, and so 10 gets assigned to `myArray[0]`. The loop variable `i` then gets incremented to 1, which is still less than `myArray.length`, and so the loop body is entered again, and 10 gets assigned to `myArray[1]`. The iterations continue until after `myArray[1999] = 10` is evaluated, at which point `i` is incremented to 2000, and the condition `i < myArray.length` becomes false, ending the `for` loop.

In the following example a `for` loop is being used to increment each element in an array of integers called `intArray`.

```
for (int i = 0; i < intArray.length; i++)
{
    intArray[i] = intArray[i] + 1;
}
```

Recall that the expression `myArray.length` will evaluate to the length of the array, which here is 2000.

SAQ 13

Using paper and pencil, write the code to declare and create an array called `hoursWorked` that can hold 52 double values, and initialise each component to 40.0.

ANSWER.....

```
double[] hoursWorked = new double[52];
for (int i = 0; i < hoursWorked.length; i++)
{
    hoursWorked[i] = 40.0;
}
```

SAQ 14

What effect does the following code have?

```
Frog[] frogPond = new Frog[10];
for (int i = 0; i < frogPond.length; i++)
{
    frogPond[i] = new Frog();
}
```

ANSWER.....

The first line of the code declares and creates an array variable `frogPond` that can hold references to 10 `Frog` objects. The `for` loop then assigns a new `Frog` object (using the `Frog` constructor) to the component at each index.

The foreach statement

If we want to access each element in an array in turn, we can use a `foreach` statement, which is a variant of the `for` statement.

For example, here is a segment of code that would print out, in turn, each of the strings in the array, referenced by `herbs`:

```
String[] herbs = {"basil", "rosemary", "thyme"};
for (String herbName : herbs)
{
    System.out.println(herbName);
}
```

Here is a template for the `foreach` statement:

```
for (declaration : expression)
{
    statement block;
}
```

In the header of the `for` statement there is a declaration of a variable (local to the `for` loop) of the same type as the component type of the array to be processed (this variable exists only for the lifetime of the `foreach` statement), and an expression which is a reference to the array to be processed. On each execution of the statement block, the variable (for example `herbName`) is assigned each element of the array (for example the array `herbs`) in turn.

To illustrate, consider this array:

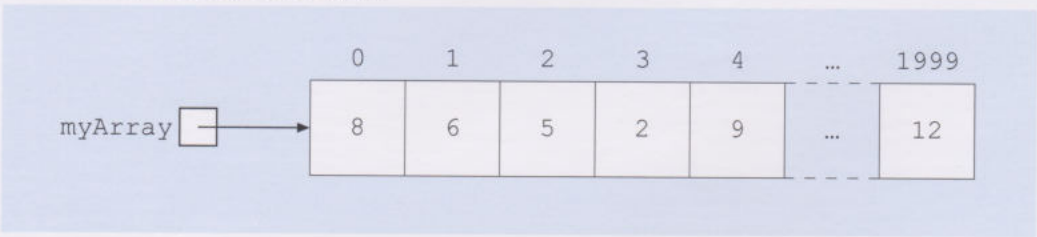


Figure 21 An array where every component is holding an integer element

Although `foreach` is not actually a Java keyword, it is an accepted computing term referring to a specific code construct hence the use of code styling.

Here a `foreach` statement is used to print out each element of `myArray`:

```
for (int anElement : myArray)
{
    System.out.println(anElement);
}
```

Within the header there are two statements separated by a colon. The first is a declaration of a variable (of the same type as the component type of the array to be processed) and the second is the name of the array to be processed. In this example `anElement` is declared as being of type `int`, and the array to be processed is `myArray`.

`anElement` is just an identifier we have chosen – it could be any other legal Java identifier.

When the `foreach` heading is first encountered during execution, the element at index 0 of `myArray` is assigned to the variable called `anElement`. After the loop body has been executed there is a check to see if there are still components to be processed; if there are, the element in the next component of `myArray` is assigned to `anElement`, if not the loop terminates.

So when the `foreach` heading is first executed, this is the situation:

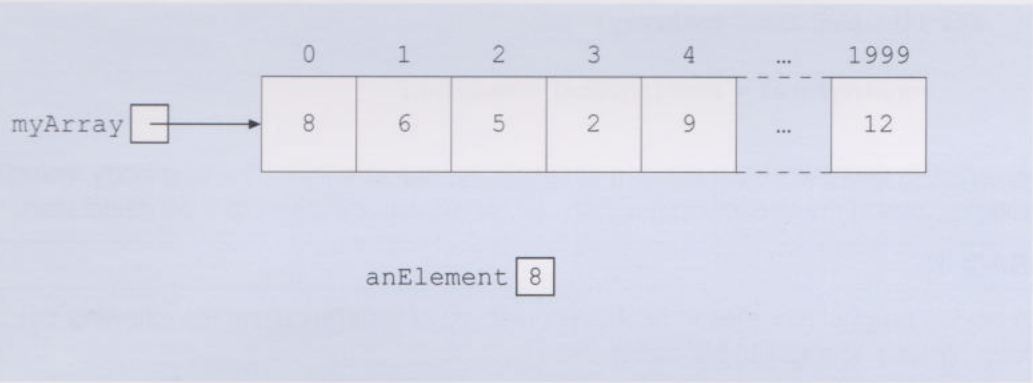


Figure 22 Entering the `foreach` loop body for the first time

The loop body is now executed, and 8 is displayed; a check is now made to see if there are still components to be processed. If there are, the element in the next component is assigned to `anElement` ready for the loop body to be executed again.

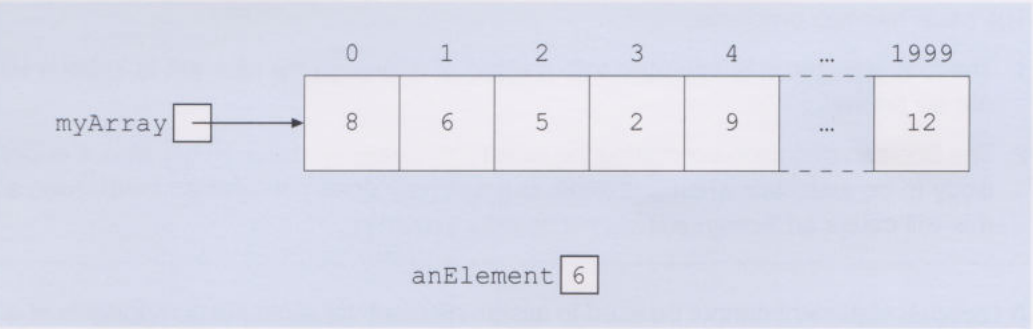


Figure 23 After the first iteration of the `foreach` loop body

The loop body is now executed, and 6 is displayed; a check is now made to see if there are still components to be processed, if there are, the element in the next component is assigned to `anElement` and the loop body is executed again, and so on.

Once the last element has been processed, the `foreach` statement terminates, and each element will have been displayed.

In chapter 10 you will come across other types of collections that cannot be iterated over using while or for loops.

It is the case that a while statement or a for statement can always be used in place of a foreach statement when iterating over arrays. For example, we could have processed myArray in exactly the same way using the code:

```
for (int i = 0; i < myArray.length; i++)
{
    System.out.println(myArray[i]);
}
```

However, the foreach loop has the practical advantage that it ensures that every component is processed, and there can be no attempt to process beyond the end of the array.

SAQ 15

Write the code to find the running total of the int values in an array called intArray.

ANSWER.....

```
int runningTotal = 0;
for (int eachInt : intArray)
{
    runningTotal = runningTotal + eachInt;
}
```

eachInt is assigned each element in intArray, one at a time. The loop body maintains a running total in the variable runningTotal, which was initialised to 0 on declaration.

SAQ 16

A novice programmer tries to print every element of an array using the following for loop. What is wrong with his code?

```
for (int i = 1; i <= myArray.length; i++)
{
    System.out.println(myArray[i]);
}
```

ANSWER.....

The code has two problems.

- 1. The loop variable i is initialised to 1 instead of 0, and so the element at index 0 will not be printed.
- 2. The Boolean condition controlling the loop (i <= myArray.length) will allow the loop body to be executed when i is 2000. But myArray does not have an index 2000 so this will cause an ArrayIndexOutOfBoundsException exception.

A foreach statement cannot be used to assign different values to the components of an array. For example, we cannot use a foreach statement to initialise an array – the following code does compile but *does not work as intended* because assigning a value to anInt is not the same as assigning a value to a component of the array:

```
for (int anInt : myArray)
{
    anInt = 10;
}
```

anInt is a variable that is local to the foreach statement. Each time the statement block executes anInt is automatically assigned the value of a component of myArray. If we

then assign another value to anInt, for example 10, this will have no effect on the component of myArray.

However, when an array's components holds references to objects, on each execution of a foreach's statement block, its local variable is automatically assigned each object in the array in turn. Therefore you can send a message to each object by using the foreach loop's local variable as the receiver. For example, consider this array, whose components are references to Frog objects:

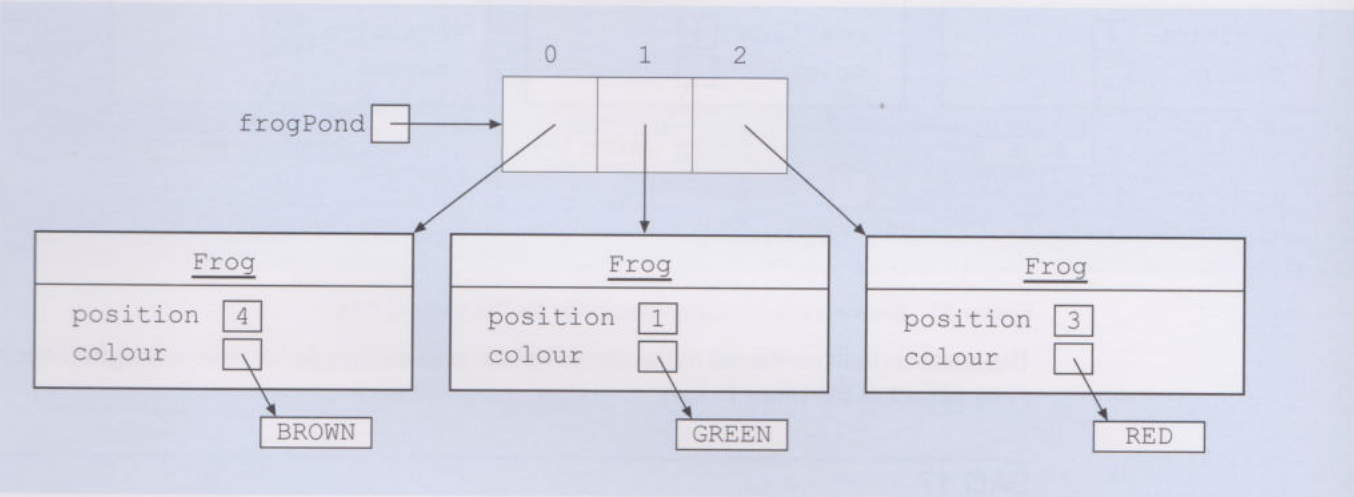


Figure 24 An array whose components hold references to Frog objects

We can use a `foreach` statement to change the colour of each `Frog` object to blue:

```
for (Frog eachFrog : frogPond)
{
    eachFrog.setColour(OUColour.BLUE);
}
```

On the first iteration the variable `eachFrog` is assigned the object at index 0, and so the situation can be depicted as follows.

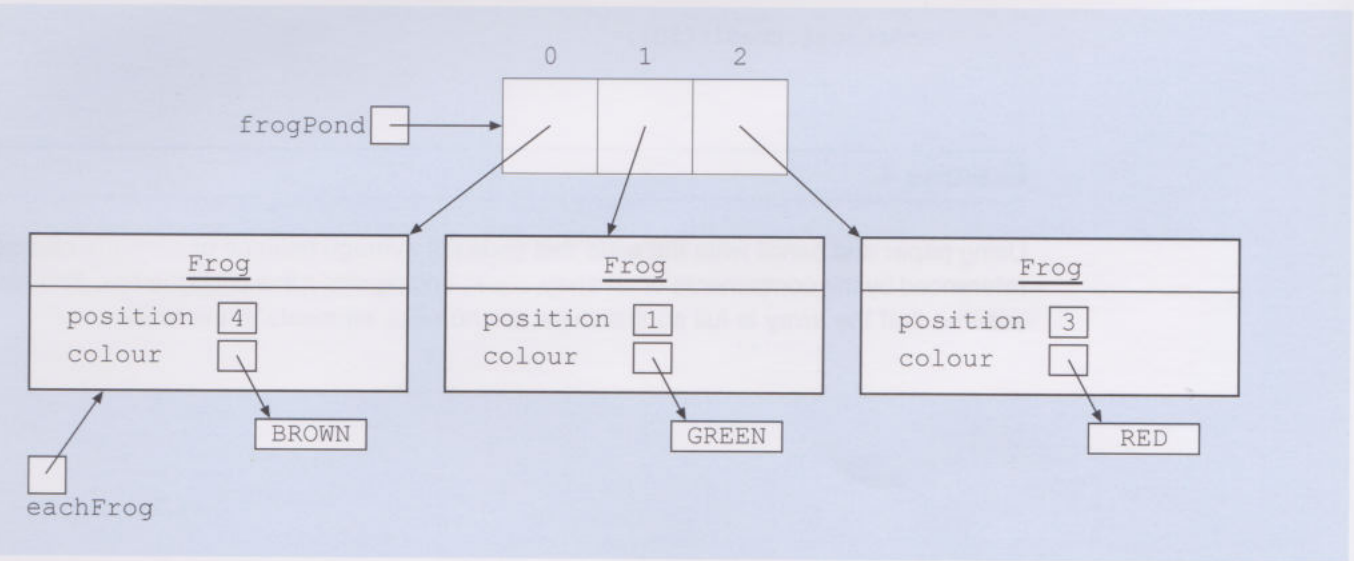


Figure 25 Entering the foreach loop body for the first time

Now when the loop body is entered `eachFrog.setColour(OUColour.BLUE)` is executed and causes the `Frog` object referenced by `eachFrog` to have its colour set to blue. The situation as the loop is entered a second time is as follows.

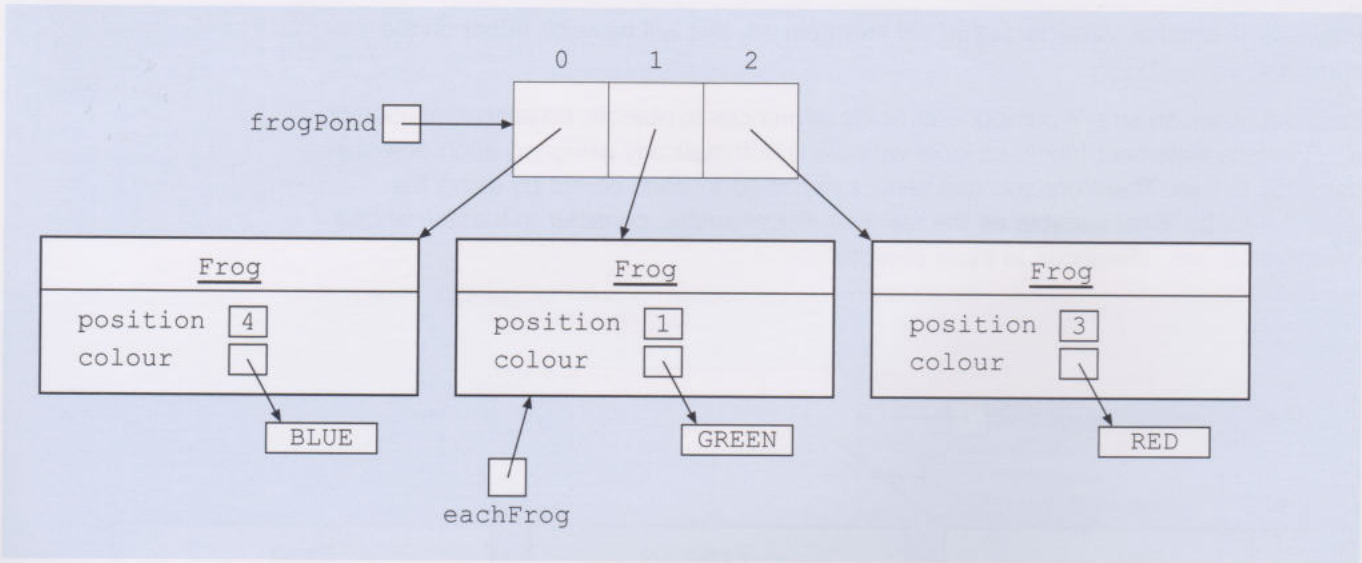


Figure 26 Entering the `foreach` loop body for the second time

The iteration then continues in the same fashion until `eachFrog` has been assigned each `Frog` object in the array in turn.

SAQ 17

An array called `bank` contains `Account` objects. The balance of each `Account` object in the array is to be increased by £50.

- (a) Can you do this processing using a `foreach` statement? Justify your answer.
- (b) Write the code to process the array in this way.

ANSWER.....

- (a) Yes. The `foreach` statement will leave the same objects in the array, but the state of the objects in the array can be changed by the `foreach` statement.

```
(b) for (Account anAccount : bank)
{
    anAccount.credit(50);
}
```

Exercise 1

Using paper and pencil write the code that finds the average balance of `Account` objects referenced by the components of an array, `bank`, and reports it in a dialogue box. You can assume that the array is full so that there are no `null` elements in the array.

Solution.....

```
double total = 0;
double average = 0;
for (Account eachAccount : bank)
{
    total = total + eachAccount.getBalance();
}
average = (total / bank.length);
OUDialog.alert("The average balance is " + average);
```

It is important to initialise `total` to 0 before the `foreach` loop as it is used in the calculation within the loop. The variable `average` has also been initialised as a matter of good practice (it is best never to rely on default initialisations). The `foreach` statement assigns each `Account` object in the array to the variable `eachAccount` in turn, and the loop body uses `eachAccount` as a receiver for a `getBalance()` message. The value returned by `getBalance()` is added to a running total. In the calculation for average the answer is a double because although `bank.length` is an `int`, `total` has been declared as a double, so the result of the division is a value of type double. In the final line the concatenation operator, `+`, converts the value of `average` to a string and the result is displayed in a dialogue box, as required.

2.2 Sub-array processing

Sub-array processing means that only part of the array (a contiguous set of components) needs to be processed. If a sub-array needs to be processed, a `foreach` statement cannot be used because it always processes *every* component, so instead we have to use either a `for` or a `while` loop.

For example, suppose that an array called `weeklyTakings` contains 52 double values to reflect the weekly takings of a shop, and you wanted to know the total takings for the first half of the year. This would involve processing only the first 26 components of the array. A `for` loop could be used as follows:

```
double total = 0;
for (int i = 0; i < 26; i++)
{
    total = total + weeklyTakings[i];
}
```

The last value of `i` should be 25, as this is the index at which the 26th element of the array is located.

SAQ 18

Write a loop fragment to calculate the total takings for the second half of the year, given the array `weeklyTakings`.

ANSWER.....

```
double total = 0;
for (int i = 26; i < 52; i++)
{
    total = total + weeklyTakings[i];
}
```

Processing arrays that are not full

Another example where sub-array processing may be necessary is if an array is not totally filled with meaningful data. In these circumstances we use the effective length of the array (which must have been calculated by the programmer as the array was being populated) as a count control variable to process the meaningful components using a `for` loop:

```
int effectiveLength;
String[] roster = new String[6];
effectiveLength = 0;
roster[0] = "Sid";
effectiveLength = effectiveLength + 1;
roster[1] = "Jill";
effectiveLength = effectiveLength + 1;
for (int i = 0; i < effectiveLength; i++)
{
    System.out.println(roster[i]);
}
```

Although this situation does occur in practice, you should note that arrays are **fixed-size collections**. If the exact number of elements needed to be stored in the array is not known in advance there may be more appropriate collection classes that can be used. Some of these will be explored in *Unit 10* and *Unit 11*.

Searching for a value in an array

If we want to know if a particular value exists in an array we could search for it by iterating over every element in the array. However, it would be more efficient to stop as soon as we had found it. In a case like this we do not know ahead of time how many iterations we will need in our loop, and so it is necessary to use a `while` statement. We need to continue searching while:

- 1 we have not found the value we are looking for, and
- 2 there are still array components left to search.

These two conditions lead to a `while` statement heading that might look like this:

```
while ((i < anArray.length) && (!found))
```

where `i` is an `int` index counter that has been initialised to 0 and `found` is a boolean flag that has been initialised to `false`.

Within the body of the loop we need to look at the element at each index in turn and compare it with what we are looking for. If the value is found, `true` is assigned to the variable `found`, if it is not, the index counter `i` is incremented. If `found` becomes `true`, or if the last component of the array is processed, the loop condition will evaluate to `false`, stopping the search.

Here is the complete code assuming that the array `anArray` contains primitive values, and the value being searched for is held in `searchValue`:

```
int i = 0;
boolean found = false;
while ((i < anArray.length) && (!found))
{
    if (anArray[i] == searchValue)
    {
        found = true;
    }
    else
    {
        i++;
    }
}
```

We would use the message `equals()` rather than `==` if we were comparing the state of two objects.

SAQ 19

Suppose that `searchValue` is found in the array. Does the programmer know where in the array it was found once the loop has exited?

ANSWER.....
Yes! The variable `i` will hold the last index processed (the one where the value was found). Note, though, that this code will find only the first occurrence of the value in the array, there may be others.

2.3 Inserting an element into a sorted array

If the elements in an array have been added in a particular order you may want to insert an element at a specific index (rather than at the next available unfilled component). For example, given the following sorted array,

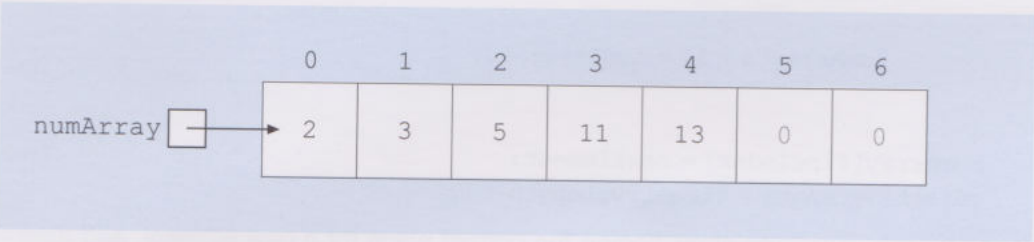


Figure 27 A sorted array

you may wish to insert a new element in such a way as to keep the elements in ascending order. To do this, first the correct index for the insertion needs to be found, which means that we need to compare the new element with the other meaningful integers in the array, one at a time. Either we will eventually find an integer that is bigger, in which case we know that the new element needs to be inserted there and the elements from this index onwards need to be moved up by one to make space for it, or we will not find a bigger integer, and we know that our new element has to be stored at the very end of the array, after the current last element.

There are many variations on this code, but in this version we assume that the calculated effective length of the array contains the number of meaningful elements (so for Figure 27, this is 5), which is less than the length of the array (there must be room to insert a new element):

```
boolean found = false;
int findIndex = 0;
while ((findIndex < effectiveLength) && (!found))
{
    if (newElement > numArray[findIndex])
    {
        findIndex++;
    }
    else
    {
        found = true;
    }
}
```

There are two possibilities after the execution of this code.

- 1 `found` could hold `true`, which means that `findIndex` holds the index where the new element needs to be inserted. In this case all of the elements of the array from `findIndex` upwards need to be shifted one place to the right (in order to make room for the new element to be inserted at `findIndex`). This shift to the right has to be done starting from the end of the meaningful array and working back to `findIndex` because otherwise elements will be overwritten in the process of the shift.
- 2 `found` could hold `false`, which means that `findIndex` holds a number one more than the index of last meaningful element. In this case the new element is larger than any element currently in the array, and so it should simply be inserted at `findIndex`.

Here is one version of code that will insert the new element in the right place in either case:

```
if (found)
{
    for (int i = effectiveLength - 1; i >= findIndex; i--)
    {
        numArray[i + 1] = numArray[i];
    }
}
numArray[findIndex] = newElement;
effectiveLength = effectiveLength + 1;
```

If you have a very large array, finding the correct place for a new element, and then moving other elements to accommodate it, is costly in terms of processor and programming time. If you need a collection to be sorted it might be better to consider using one of the purpose built classes for sorted collections that you will see in *Unit 11*.

ACTIVITY 3

Launch BlueJ and open project `Unit9_Project_1`. Then from the Tools menu select the `OJWorkspace`. Open the file called `CodeForActivity3.txt`.

In the following activities `intArray` is an array that holds integers. You may want to create the array `{10, 5, 7, 2, 9, 8, 1, 12}` to test your code, but your code should work on any full array of integers.

Make sure that the Show results checkbox is checked and then enter, select and execute statements to do each of the following (inspect the variables when appropriate to check your code works as expected).

- 1 Print the textual representation of each element in `intArray`, one to a line, to the Display Pane.
- 2 Display the total of all the elements of `intArray` in a dialogue box.
- 3 Find all the elements in `intArray` that are greater than 8 and replace each of them by 0.
- 4 Create a second array, `intArrayCopy`, that is an exact copy of `intArray`.

Make sure that the Show results checkbox is checked and then enter, select and execute statements to do each of the following (you should inspect variables where appropriate):

- 5 Prompt the user to enter the desired length of an array (with a default option of 10), and then create an `int` array called `myArray` of this length. The user should then be repeatedly invited to enter a value (with a default option of 0) to be stored in the array. The user should be prompted to enter exactly enough values to fill the components of the array.
- 6 Check if two `int` arrays, `anArray` and `anotherArray`, are identical arrays (that is they contain exactly the same elements in the same order). Your code should use a variable named `identical` that should hold either `true` if they are identical or `false` otherwise. You should test your code on each of the following pairs of arrays.

```
int[] anArray = {7, 6, 9, 2, 6, 10, 8};
int[] anotherArray = {7, 6, 9, 2, 6, 10, 8};
```

```
int[] anArray = {7, 6, 9, 2, 6, 10, 8};
int[] anotherArray = {7, 6, 9, 1, 6, 10, 8};
```

```
int[] anArray = {7, 6, 9, 2, 6};
int[] anotherArray = {7, 6, 9, 2, 6, 10, 8};
```

```
int[] anArray = {};
int[] anotherArray = {};
```

- 7 Find the first occurrence of a red frog in an array of `Frog` objects called `frogPond`. Display a dialogue box that either gives the index at which the first red frog is found, or announces that no red frogs exist in the pond.

You should test your code on ponds that contain:

- ▶ no red frogs;
- ▶ exactly one red frog;
- ▶ more than one red frog.

Hint: First create the `frogPond` array with the components holding references to newly initialised frogs (so none of them are red). You can do this by evaluating the code:

```
Frog[] frogPond = {new Frog(), new Frog(), new Frog()};
```

Once you have tested this, send a message to change the colour of one of the frogs to red, and test again. Finally change the colour of another frog to red and test again.

DISCUSSION OF ACTIVITY 3

```
1  for (int anInt : intArray)
    {
        System.out.println(anInt);
    }
```

Here `anInt` is assigned, one at a time, the value of each element of `intArray`.

```
2  int sum = 0;
    for (int anInt : intArray)
    {
        sum = sum + anInt;
    }
    OUDialog.alert("sum is " + sum);
```

The variable `sum` must be initialised to 0 before the `foreach` statement. A running total is calculated within the loop body. Using the suggested test data, `sum` should hold the value 54.

```
3  for (int i = 0; i < intArray.length; i++)
    {
        if (intArray[i] > 8)
        {
            intArray[i] = 0;
        }
    }
```

Here a `for` loop is needed (rather than a `foreach` loop) because the processing requires the primitive values held in the array components to be changed. Each array element in turn is compared to 8. If it is greater than 8 the array element is overwritten by 0, i.e. the component at the current index is assigned 0.

```
4  int[] intArrayCopy = new int[intArray.length];
    for (int i = 0; i < intArray.length; i++)
    {
        intArrayCopy[i] = intArray[i];
    }
```

The first line creates a new array, `intArrayCopy`, which is the same length as the existing `intArray`. Within the loop body each element of `intArray` is assigned to the corresponding component of `intArrayCopy`.

Notice that here we have to use a `for` loop (rather than a `foreach` statement) because we need to change the existing elements of `intArrayCopy` (which have been initialised to the default value of 0).

You might have tried `intArrayCopy = intArray` but this does not create a new array, rather it provides a second reference, `intArrayCopy`, for the array referenced by `intArray`. You can verify this by evaluating `intArrayCopy == intArray` which will return `true`, showing that they reference the same object.

```
5  String desiredLength;
    String input; desiredLength =
        OUDialog.request("How many integers do you wish to store?", "10");
    int[] myArray = new int[Integer.parseInt(desiredLength)];
    for (int i = 0; i < myArray.length; i++)
    {
        input = OUDialog.request("Please enter an integer", "0");
        myArray[i] = Integer.parseInt(input);
    }
```


Did you remember that the results from the dialogue box needed to be converted from a `String` to an `int`? In our solution we used a temporary variable, but the conversion and assignment could have been done in one statement as follows (assuming that `input` is declared as an `int`):

```
input = Integer.parseInt(OUDialog.request("Please enter an integer", "0"));
```

- 6 Here is one possible solution. Notice that checking that the two arrays have the same length is a sensible first step (because there is no need to proceed if they do not). The `while` loop exits if two elements with the same index are not identical (the boolean flag becomes `false`), or when the last component has been processed (`i` exceeds the length of the array).

```
boolean identical = true;
int i = 0;
if (anArray.length == anotherArray.length)
{
    while ((i < anArray.length) && (identical))
    {
        if (anArray[i] != anotherArray[i])
        {
            identical = false;
        }
        i++;
    }
}
else
{
    identical = false;
}
```

When you inspect `identical` after each run of the given test data it should hold `true`, `false`, `false` and `true`, respectively.

- 7 Again we use a `while` loop to iterate through the array until either a red frog is found, or there are no more components.

```
int i = 0;
boolean found = false;
while ((!found) && (i < frogPond.length))
{
    if (frogPond[i].getColour().equals(OUColour.RED))
    {
        found = true;
    }
    else
    {
        i++;
    }
}
if (found)
{
    OUDialog.alert("Red frog found at index " + i);
}
else
{
    OUDialog.alert("No red frogs found in this pond");
}
```

2.4 Two-dimensional arrays

The arrays we have used so far have been one-dimensional – they represent items in a list or a single sequence. A **two-dimensional array** can be used to represent a two-dimensional table of values, like this one:

	0	1	2	3	4
0	4	3	1	5	3
1	9	5	3	8	2
2	4	2	9	4	3

Figure 28 A two-dimensional array

The above table is an array of length 3, each component of which references an array of length 5:

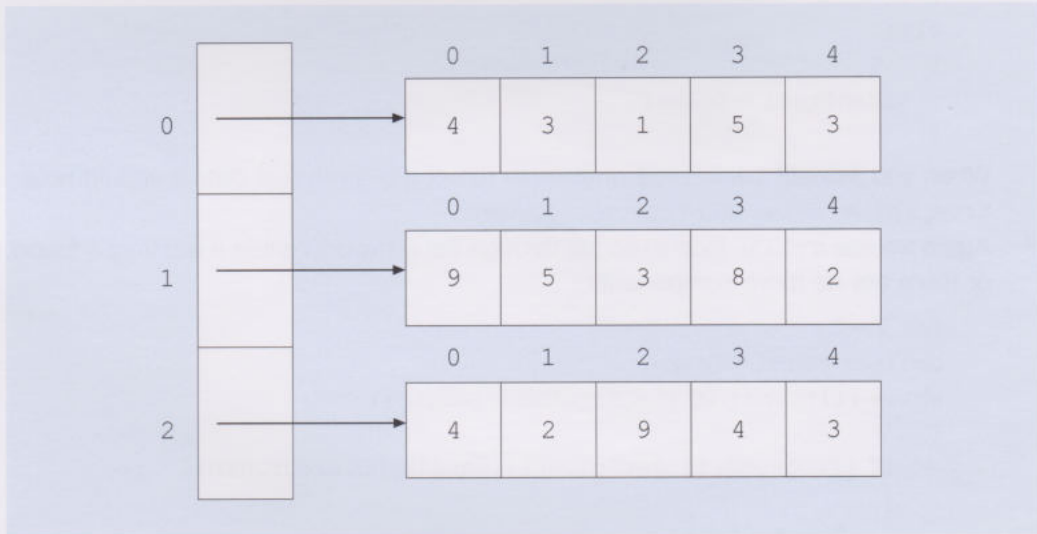


Figure 29 A two-dimensional array

Here is the code to declare a variable to reference a two-dimensional array to hold `int` elements, and assign a newly created two dimensional array to that variable.

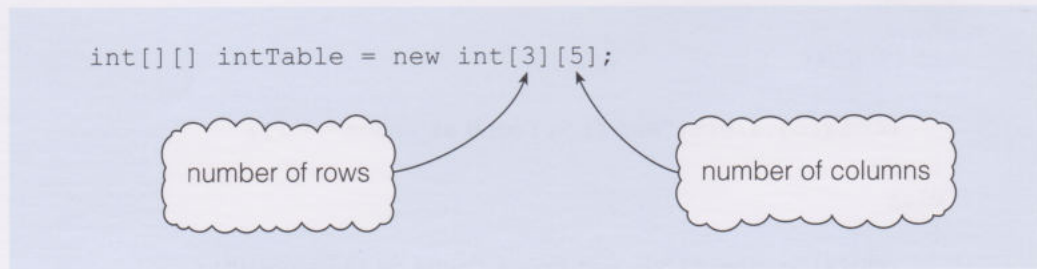


Figure 30 Creating a two-dimensional array of integers

To put an integer in a particular component we can think of giving a kind of 'grid reference', so,

```
intTable[1][4] = 8;
```

places an 8 in row 1 of column 4.

The diagram shows a 3x5 grid representing the `intTable` array. The columns are indexed 0 to 4, and the rows are indexed 0 to 2. The cell at row 1, column 4 is highlighted with a blue border and contains the value 8. An arrow points from the label `intTable` to the first column of the grid.

	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0	8
2	0	0	0	0	0

Figure 31 The component `intTable[1][4]`

Suppose that we now wanted to put 6 in every column in row 2. We could use a `for` loop to do this:

```
for (int i = 0; i < 5; i++)  
{  
    intTable[2][i] = 6;  
}
```

In this code we have 'fixed' the row index to 2, and allowed the column counter, `i`, to go from 0 to 4, assigning 6 to each of the 5 elements in row 2.

SAQ 20

Consider the situation where all the elements in `intTable` are 0, as they would be when the array is first created.

The diagram shows a 3x5 grid representing the `intTable` array. All cells in the grid contain the value 0. An arrow points from the label `intTable` to the first column of the grid.

	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0	0
2	0	0	0	0	0

Figure 32 The array `intTable` when it is first created

(a) What will `intTable` contain after the following for loop has been executed?

```
for (int i = 0; i < 5; i++)  
{  
    intTable[2][i] = 7;  
}
```

(b) Again, with all the elements being 0, what will `intTable` contain after the following for loop has been executed?

```
for (int i = 0; i < 3; i++)  
{  
    intTable[i][1] = 7;  
}
```

ANSWER.....

(a) In the code, the *row* number is fixed at 2, and the column count goes from 0 to 4, so 7 is placed in each column of row 2.

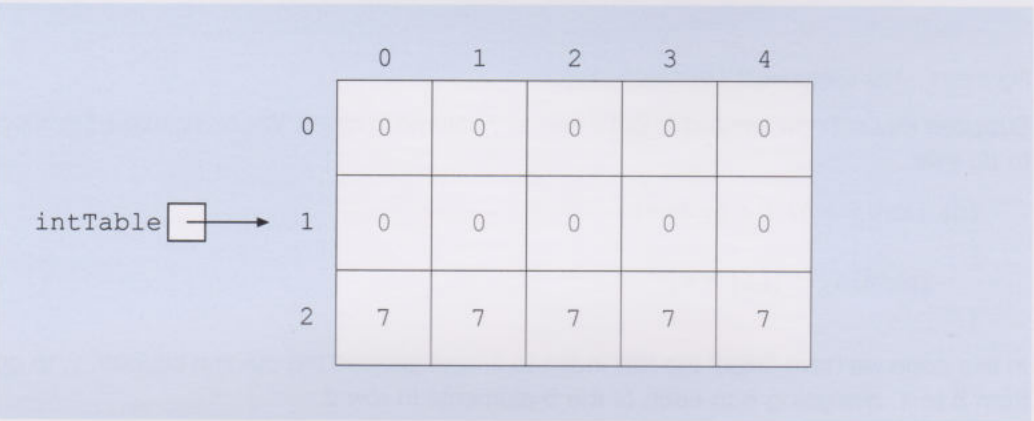


Figure 33 `intArray` after 7 has been placed in each column of row 2

(b) In the code, the *column* number is fixed at 1, and the row number iterates from 0 to 2, so 7 is placed in each row of column 1.

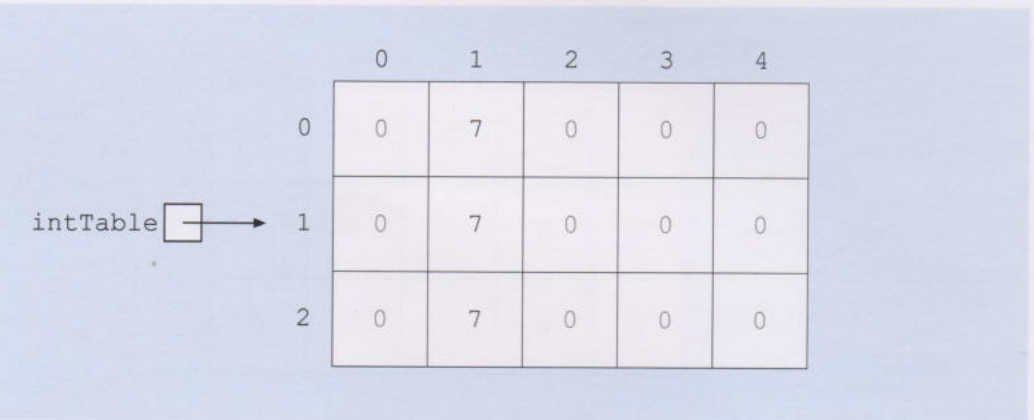
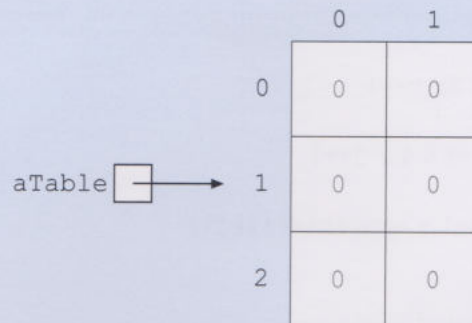


Figure 34 `intArray` after 7 has been placed in each row of column 1

Processing a complete table

The following code creates a table with 3 rows and 2 columns:

```
int[][] aTable = new int[3][2];
```



	0	1
0	0	0
1	0	0
2	0	0

Figure 35 A newly created two-dimensional array

Now consider this code;

```
for (int i = 0; i < 3; i++)
{
    for (int j = 0; j < 2; j++)
    {
        aTable[i][j] = 10;
    }
}
```

We can see that the outer `for` loop, with a loop control variable `i` is controlling the rows. The inner `for` loop, with a loop control variable `j` is controlling the columns.

When the outer loop is first encountered, `i` is set to 0, and the outer loop body is entered. The inner `for` loop then iterates twice, for `j = 0` and `j = 1`. Since `i` remains as 0, the effect is that each component in row 0 is initialised to 10. Once `j` has been incremented to 2, the inner loop condition is `false`, and control is returned to the outer loop. The outer loop control variable, `i`, is incremented to 1, and as the outer loop condition is still `true`, the loop body is entered, and the inner loop again iterates twice for `j = 0` and `j = 1`. This time because `i` remains as 1, it is each component in row 1 that is initialised to 10. When `j` reaches 2 the inner loop exits, and control returns to the outer loop, where `i` is incremented to 2. The inner loop then iterates twice, initialising each component of row 2 to 10. When control returns to the outer loop `i` is incremented to 3, causing the outer loop to exit because the outer loop condition becomes `false`.

Tracing through a nested loop like this is difficult, but the way the code works should be apparent. The outer loop controls which row is being processed, and the inner loop processes each column in the current row. At the end of the looping process we have processed each column within each row, and the whole two-dimensional array has been processed.

SAQ 21

With paper and pencil write code to find the total of all the elements in a two-dimensional array named numTable, which is defined as follows.

```
double[][] numTable = new double[9][4];
```

ANSWER.....

```
double total = 0;
for (int i = 0; i < 9; i++)
{
    for (int j = 0; j < 4 ; j++)
    {
        total = total + numTable[i][j];
    }
}
```

For each row in turn (controlled by the outer loop), the inner loop processes each column.

Declaring and creating a two-dimensional array using literals

Because Java treats a two-dimensional array as an array that has components that are also arrays, we can create a ready initialised two-dimensional array using the syntax for a literal array we saw earlier for one-dimensional arrays:

```
int[][] aTable = {{1, 3, 5}, {7, 9, 11}};
```

which represents a table with 2 rows and 3 columns, as shown in the next figure. Note that:

```
int[][] aTable;
aTable = new int[][]{{1, 3, 5}, {7, 9, 11}};
```

will also work.

	0	1	2
0	1	3	5
1	7	9	11

Figure 36 A two-dimensional array

ACTIVITY 4

Launch BlueJ and open project Unit9_Project_1.

Look at this table (Figure 37), which is known as Dürer's magic square because it appears in an engraving called *Melancholia* that Dürer made in 1514 (notice the 15 and 14 in the middle of the last row).

16	3	2	13
5	10	11	8
9	6	7	12
4	15	14	1

Figure 37 A magic square

The rows and columns of Dürer's magic square each add up to 34, as do the diagonals. Each corner quadrant, and the middle square also add up to 34.

16	3	2	13
5	10	11	8
9	6	7	12
4	15	14	1

16	3	2	13
5	10	11	8
9	6	7	12
4	15	14	1

Figure 38 The quadrants and centre square of the magic square

Make sure that the Show results checkbox is checked and then enter, select and execute statements to do each of the following.

- 1 Using the literal array syntax create a new array referenced by a variable called `intTable` to represent Dürer's magic square.
- 2 Report in a dialogue box the sum of all the elements in `intTable`.
- 3 Report in separate dialogue boxes the sum of each row of `intTable` (giving the appropriate row number each time).
- 4 Report in separate dialogue boxes the sum each column of `intTable` (giving the appropriate column number each time).

If you have time you may also want to check the two diagonals, and the five individual squares.

DISCUSSION OF ACTIVITY 4

```

1  int[][] intTable = {{16,3,2,13}, {5,10,11,8}, {9,6,7,12}, {4,15,14,1}};
2  int sum = 0;
   for (int i = 0; i < 4; i++)
   {
       for (int j = 0; j < 4; j++)
       {
           sum = sum + intTable[i][j];
       }
   }
   OUDialog.alert("The sum of all elements is " + sum);

```

Here the outer loop, controlled by *i*, processes each row in turn. For each row in turn the inner loop, controlled by *j*, processes each column. Hence the sum is calculated for each column in turn and for each row in turn, giving a total that is reported outside the nested loop.

```

3  int sum = 0;
   for (int i = 0; i < 4; i++)
   {
       for (int j = 0; j < 4; j++)
       {
           sum = sum + intTable[i][j];
       }
       OUDialog.alert("Row " + i + " has a total of " + sum);
       sum = 0;
   }

```

Here the outer loop, controlled by *i*, processes each row in turn. For each row in turn, the inner loop, controlled by *j*, processes each column. However, we want the sum to be displayed at the end of each row processed, and so *sum* needs to be reset to 0 before the next row is processed. Therefore, statements to perform these tasks are placed before the end of the outer loop body.

```

4  int sum = 0;
   for (int j = 0; j < 4; j++)
   {
       for (int i = 0; i < 4; i++)
       {
           sum = sum + intTable[i][j];
       }
       OUDialog.alert("Column " + j + " has a total of " + sum);
       sum = 0;
   }

```

This is similar code to the previous answer, but here the outer loop, controlled by *j*, processes each column in turn. For each column in turn, the inner loop, controlled by *i* processes each row. However, we want the sum to be displayed at the end of each column processed, so *sum* needs to be reset to 0 before the next column is processed. Therefore statements to perform these tasks are placed before the end of the outer loop body.

For your interest, the code to check the sum of the diagonals and for the top left-hand quadrant is shown below.

Diagonal down (from left to right):

```
int sum = 0;
for (int k = 0; k < 4; k++)
{
    sum = sum + intTable[k][k];
}
OUDialog.alert("The sum of the diagonal down elements is " + sum);
```

Diagonal down (from right to left):

```
int sum = 0;
for (int k = 0; k < 4; k++)
{
    sum = sum + intTable[k][3 - k];
}
OUDialog.alert("The sum of the diagonal up elements is " + sum);
```

Top left-hand quadrant:

```
int sum = 0;
for (int i = 0; i < 2; i++)
{
    for (int j = 0; j < 2; j++)
    {
        sum = sum + intTable[i][j];
    }
}
OUDialog.alert("The sum of the top left quadrant is " + sum);
```

Multi-dimensional arrays

Although we have talked only about two-dimensional arrays, Java allows three- or four- (or more) dimensional arrays. In each case the array is considered to be an array of arrays. For example, a three-dimensional array is considered to be an array whose components reference two-dimensional arrays.

2.5 java.util.Arrays

Now you have worked through lots of exercises using arrays we have a confession to make. Java has a built-in library class called `java.util.Arrays` that does some of the routine tasks that we often need to do when working with arrays. In the activity that follows you will explore some of the static methods within this utility class. These methods are all overloaded so that there are methods with the same name for handling arrays of primitive values or objects.

ACTIVITY 5

Launch BlueJ and open project Unit9_Project_1. Then from the Tools menu open the OUWorkspace. Open the file called CodeForActivity5.txt, select and execute the code:

```
int[] firstIntArray = {9, 3, 7, 2, 8, 4, 10, 1, 5, 6};
int[] secondIntArray = {9, 3, 7, 2, 8, 4, 10, 1, 5, 6};
String[] firstStringArray = {"Bob", "Anne", "Alan", "Mani"};
String[] secondStringArray = {"Mani", "Alan", "Bob", "Anne"};
int[][] firstTable = {{1, 3, 2}, {5, 9, 4}, {3, 7, 10}, {8, 1, 6}};
int[][] secondTable = {{1, 3, 2}, {5, 9, 4}, {3, 7, 10}, {8, 1, 6}};
int[][] thirdTable = {{3, 1, 2}, {5, 9, 4}, {3, 7, 10}, {8, 1, 6}};
```

Select and execute each of the following statements, one at a time. Describe the effect of each statement. If you are not sure of the effect, create additional arrays to experiment further.

- 1 `Arrays.toString(firstIntArray);`
`Arrays.toString(firstStringArray);`
- 2 `Arrays.deepToString(secondTable);`
- 3 `Arrays.equals(firstIntArray, secondIntArray);`
`Arrays.equals(firstStringArray, secondStringArray);`
- 4 `Arrays.deepEquals(firstTable, secondTable);`
- 5 `Arrays.sort(firstIntArray);`
`Arrays.sort(firstStringArray);`
- 6 `Arrays.fill(firstIntArray, 0);`
- 7 `Arrays.fill(secondIntArray, 2, 8, 100);`
- 8 From BlueJ's Help menu choose Java Class Libraries. When the browser opens, select, from the left-hand frame, the `Arrays` class. Find a static method of `Arrays` that can be used to determine whether a particular integer is in a sorted array. Read the description of this method carefully. Write the statements you would execute to first sort `secondIntArray`, and then determine whether the integer 9 is present in the sorted array. Predict what will be returned. Test your code.
- 9 Next, still using the Javadoc, find a static method of `Arrays` that can be used to determine whether a particular object is in a sorted array. Write the statement you would execute to determine whether "Anne" is present in `firstStringArray`, and predict what will be returned. Finally, write the statement you would execute to determine whether "Bill" is in `firstStringArray`, and predict what will be returned. (Note that `firstStringArray` should already be sorted from part 5 above.) Test your code.

DISCUSSION OF ACTIVITY 5

- 1 The following is displayed in the Dialogue Pane when the first statement is executed:

```
"[9, 3, 7, 2, 8, 4, 10, 1, 5, 6]"
```

and the second statement displays the following.

```
"[Bob, Anne, Alan, Mani]"
```

The effect of `Arrays.toString()` is to return a string representation of the contents of the array 'topped' and 'tailed' with square brackets.

- 2 The following is displayed.

```
"[[1, 3, 2], [5, 9, 4], [3, 7, 10], [8, 1, 6]]"
```

The effect of `Arrays.deepToString()` is to return a string representation of the contents of the array given as the method's argument, including the contents of any of the arrays within the argument array.

- 3 The first expression returns `true` and the second returns `false`. The effect of `Arrays.equals()` is to return `true` if the two arrays given as the method's argument are equal (they contain exactly the same elements in the same order) and `false` otherwise. If the component type of the arrays is some primitive type, then corresponding pairs of elements in the two arrays are compared by value using `==`. If the component type of the arrays is some class type, then corresponding pairs of elements in the two arrays are compared using `equals()`. This means that, unless the `equals()` method inherited from `Object` has been overridden by that class (or one of its superclasses), the elements in the arrays will be compared for identity rather than state.
- 4 The statement returns `true`. The effect of `Arrays.deepEquals()` is to return `true` if the two arrays given as the method's arguments are deeply equal and `false` otherwise. Deeply equal means that if the arrays themselves contain arrays, the pairs of corresponding nested arrays must also be equal.
- 5 The components of `firstIntArray` now hold the integer elements in the following order:

```
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
```

and the components of `firstStringArray` now hold the string elements in the following order:

```
{"Alan", "Anne", "Bob", "Mani"}.
```

The effect of `Arrays.sort()` is to sort the elements of the argument array into ascending order (as long as the elements can be compared). The `Arrays` class also has another version of the `sort()` method that allows part of an array to be sorted. For example the code `Arrays.sort(intArray, 4, 8)` would sort the elements at indexes 4 to 7, but leave the others unchanged. Note it would not sort up to index 8 because the third argument to this method is exclusive whereas the second argument is inclusive.

- 6 The components of `firstIntArray` now hold the following.

```
{0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
```

The effect of `Arrays.fill()` is to fill the components of the argument array with the given value. This method can be used with any type of literal value as the second argument (as long as it matches the component type of the array).

- 7 The components of `secondIntArray` now hold the following elements, in the following order.

```
{9, 3, 100, 100, 100, 100, 100, 100, 5, 6}
```

The effect of this version of `Arrays.fill()` is to fill just part of an array. The first argument is the array, the next two `int` values give the starting and ending indexes, and the fourth argument gives the value to be assigned to the components within those indexes. The array is filled from the start index to the end index -1 (that is, in this case, from index 2 to index 7).

- 8 The static method signature is `binarySearch(int[], int)`. A binary search should only be conducted on a sorted array. The code to be executed is:

```
Arrays.sort(secondIntArray);  
Arrays.binarySearch(secondIntArray, 9);
```

This returns 3, indicating that 9 was found at index 3.

- 9 The static method signature is `binarySearch(Object[], Object)`. The two expressions to be executed are:

```
Arrays.binarySearch(firstStringArray, "Anne");  
Arrays.binarySearch(firstStringArray, "Bill");
```

The first expression returns 1, indicating that "Anne" was found at index 1. The second expression returns -3. The negative value indicates that "Bill" was not found in the array. The value of the negative `int` is given by,

$(- (\text{insertion point}) - 1)$

where the insertion point is the index at which the value should be inserted in order to maintain the sorting. So if, for example, -3 is returned this indicates that "Bill" should be inserted at index 2 (one less than the absolute value of the return value).

3 The `main()` method

So far in this course you have used BlueJ to write and adapt classes and then to compile them. You have then used the OUWorkspace to write and execute code that creates instances of those classes and then send messages to them. In this way you have made frogs dance, transferred money between bank accounts, converted currency and displayed marionettes, all by sending messages to objects in the OUWorkspace. In effect, you have used the OUWorkspace to write small programs.

In this section we will look at how programs can be written without using the OUWorkspace, or even BlueJ.

Any program that is executed outside the OUWorkspace (and in real life this is the normal situation) has to have an entry point; it must start somewhere. To define the starting point for execution, Java uses the mechanism of a `main()` method, which follows the approach used by the earlier programming language C.

To run a Java program, the Java Virtual Machine (JVM) loads a particular class that must have been previously specified. This class must contain a special static method called `main()`. The JVM then executes this `main()` method, which sets the ball rolling and everything else the program does cascades from this. To all intents and purposes, `main()` does the same job as the OUWorkspace.

Java is very strict on how the `main()` method is defined; it must have exactly this header:

```
public static void main(String[] args)
```

As stated, the purpose of the `main()` method is to provide a starting point for the execution. Therefore it must be declared as `public` because it has to be available to the world outside the class in which it is defined. It must also be declared as `static`, because `main()` is the first method to be invoked, and at this point no objects exist. It does not return anything, so its return type is `void`. You will also see that it has a single argument, which you should recognise as an array of `String` objects, called `args`. We will talk more about this argument later in this section.

3.1 Developing programs outside the BlueJ environment

It is perfectly possible to compile and execute programs without using the BlueJ environment (or any other IDE such as JBuilder or NetBeans) as long as the Java Development Kit (JDK) is available on your computer. The source code can be written and edited in any text editor (for example, Microsoft Notepad), and then compiled and executed from the command prompt window with the command prompt tools provided with the JDK. If a program is developed outside of the BlueJ environment it must include a `main()` method, so let us look a bit more closely at what a `main()` method might look like.

The command prompt window is also known as the C prompt, the shell, the cmd prompt and the command line.

Here is an example of a very simple class.

```
/**
 * Prints "Hello world" to the default output device
 */
public class Welcome
{
    /**
     * main() method
     */
    public static void main(String[] args)
    {
        welcome.start();
    }

    /**
     * Prints "Hello world" to the default output device.
     */
    public static void start()
    {
        System.out.println("Hello world");
    }
}
```

The `Welcome` class consists of two static methods, `main()` and `start()`. The code in the body of `main()` merely invokes the `start()` method. It is generally the case that the `main()` method simply 'passes the buck' by invoking another method, in this case `start()`, within which resides the code to coordinate the meaningful task. Here the meaningful task is to output the phrase 'Hello World' to the standard output device.

ACTIVITY 6

We would recommend Notepad, or any other simple text editor because word-processing packages often insert special characters, such as smart quotes, which may not be recognised by the Java compiler.

In this activity you will create, compile and run the `Welcome` program without using the BlueJ environment. For your convenience we have provided a copy of the source code in the folder `Unit9_Activities_6-8` in a file called `Welcome.txt`.

Do *not* open BlueJ, instead open this file within a text editor, such as Notepad. Do not modify it, but save it with the name `Welcome.java`. In Notepad you can do this by using `SaveAs`, and then changing the `Save as type:` field to `All Files`, before typing the filename `Welcome.java` in the `File name:` field. (Note that in Java, class source files must end in `.java` and must be named with the same name as the class they contain.)

Many text editors add their own file extension as a default, for example `.doc` for Word documents, so if you are not using Notepad ensure that the file really has been saved as `Welcome.java` (and not `Welcome.java.doc`, for example.)

You should now access the command prompt window. For your convenience we have provided an easy means to do this, and to set the correct path. Simply double-click on the file called 'command prompt' which is in the `Unit9_Activities_6-8` folder. You should find that the command prompt window opens and (depending on where you installed the M255 software) looks something like this:

```
C:\> ... M255\M255Projects\Block3\Unit9_Activities_6-8>
```


Now to compile the Welcome source file you need to type the JDK compile command, `javac`, followed by the file name and extension of your source file like this:

```
C:\> ... M255\M255Projects\Block3\Unit9_Activities_6-8>javac Welcome.java
```

The source code provided contained no syntax errors so it should compile. You will not get any feedback that this has happened in the command prompt window, but you should find that a new file called `Welcome.class` will have automatically been created for you in the same folder as your original `.java` source file. (Note that all compiled class files end with `.class` in Java.)

Now you are ready to execute your program. In the command prompt window type `java` followed by the name of the `.class` file. Note that you must *not* include the `.class` extension here.

```
C:\> ... M255\M255Projects\Block3\Unit9_Activities_6-8>java Welcome
```

Once your program has run correctly, open the file from a text editor such as Notepad and introduce a syntax error into the source code, for example, type an extra `}` at the end. Save this file, and try to compile it again. What happens? Now correct the syntax error, but delete or comment out the `main()` method in the source code before compiling it and running it again. What happens?

If you have another Java IDE installed on your computer (in addition to BlueJ), do not double-click `.java` files, as it may result in the other IDE starting up.

DISCUSSION OF ACTIVITY 6

When your program executes, `Hello world` is output to the command prompt window.

After you have introduced a syntax error, the class will not compile, and the following error message appears:

```
Welcome.java:24: 'class' or 'interface' expected
}
^
Welcome.java:25: 'class' or 'interface' expected
}
^
```

indicating that the source code does not have the correct syntax for a class or interface.

If `main()` is commented out the class will compile successfully, but attempting to execute it produces the following error message:

```
Exception in thread "main" java.lang.NoSuchMethodError: main
```

indicating that the class does not constitute a program as it does not have a `main()` method.

3.2 Arguments to the main() method

We have not yet explained the purpose of the argument to the `main()` method, which as you recall is in the form of `String[] args`. You learnt in *Unit 4* that arguments are the means by which the additional information that the method needs to do its job is made available to the method. The same applies to the `main()` method. When a program is executed from the command prompt, any additional information the program needs can be typed in as arguments at the command prompt. These arguments are then stored in the `args` array and are available to be used by the `main()` method.

Consider this amended program:

```
/**
 * Prints the name entered as arguments at the command prompt
 */
public class Welcome2
{
    /**
     * The main() method
     */
    public static void main(String[] args)
    {
        Welcome2.start(args);
    }

    /**
     * Prints names to the standard output device
     */
    public static void start(String[] names)
    {
        System.out.print("Hello");
        for (String eachName : names)
        {
            System.out.print(" " + eachName);
        }
    }
}
```

Now suppose this source code is saved as Welcome2.java and successfully compiled. It can then be executed from the command prompt, as follows:

C:\>...M255\M255Projects\Block3\Unit9_Activities_6-8>java Welcome2 John Rob Kermit

The command line arguments, John, Rob and Kermit are stored in the args array which is the argument of the main() method. So when the program is executed, args will contain the three string elements "John", "Rob" and "Kermit" at indexes 0, 1 and 2, respectively.

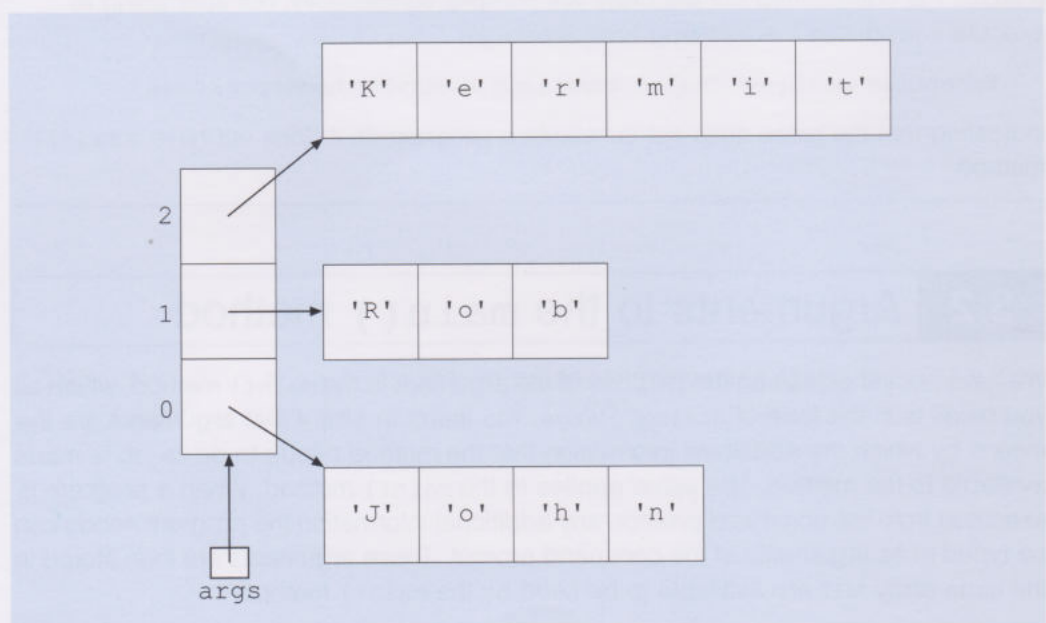


Figure 39 The args array for Activity 6

In our simple program, `args` is then passed as an argument to the `start()` method, where each of the `String` elements is printed in turn, using a `foreach` statement.

In fact, command line arguments are rarely used in Java, but the syntax requires the use of `String[] args` as the argument for every `main()` method.

ACTIVITY 7

- 1 The source code for the `Welcome2` class has been saved as `Welcome2.txt`, in the folder `Unit9_Activities_6-8`. Save this file as `Welcome2.java`, and compile and run it at the command prompt with a variety of different arguments.
- 2 Save your source file as `Welcome3.java`. Using `Welcome3.java` carefully amend the source code so that it outputs the command line arguments in reverse order. You will have to use a `for` loop to do this (rather than a `foreach`), with the loop counter initialised to the last index of the array, and being decremented after each iteration. (Note that you will need to change any references to `Welcome2` in the source code to `Welcome3`.) Compile and run this new program.

DISCUSSION OF ACTIVITY 7

- 1 You should find that `Hello` followed by the command line arguments is output.
- 2 You want to print out the last element of the array, down to the element at index 0, so the `start()` method should be modified as follows.

```
public static void start(String[] names)
{
    System.out.print("Hello");
    for (int i = names.length - 1; i >= 0; i--)
    {
        System.out.print(" " + names[i]);
    }
}
```

3.3 A complete program that uses arrays

ACTIVITY 8

In this activity you will compile and run a standalone program that grades a multiple-choice quiz. You will run the program from the command prompt with a pupil's answers being entered as command line arguments. There are four possible answers to each question, which will be stored as the strings "A", "B", "C" and "D". When it is run, the program will take 10 command line arguments and these will be used to construct an array called `pupilsTest` of the pupil's answers. The key for the test will be 'hard coded' into the program as an array called `key`. The pupil's quiz will be graded by comparing each element of `key` to the corresponding element of `pupilsTest`, and the result (as a percentage) will be displayed in a dialogue box, along with a message saying whether the pupil passed or failed (anything less than 40% is a fail). Here is the source code, which is also available in the `Unit9_Activities_6-8` folder, as a `.txt` file called `QuizMarker.txt`.

```

public class QuizMarker
{
    private static final String[] KEY = {"D","A","A","B","C","B","A","C","C","D"};
    private String[] pupilsTest;

    /**
     * Creates an instance of class QuizMarker
     * and sends it the message markQuiz()
     */

    public static void main(String[] args)
    {
        QuizMarker marker = new QuizMarker(args);
        marker.markQuiz();
    }

    /**
     * The method that controls the program.
     */
    public void markQuiz()
    {
        // the return value from mark() is used as
        // the argument for reportResults() message
        this.reportResults(this.mark());
    }

    /**
     * Constructor for objects of class QuizMarker. It will initialise
     * the array pupilsTest to be identical to args (the elements of
     * arg come from the command line arguments).
     */
    public QuizMarker(String[] args)
    {
        this.pupilsTest = args;
    }

    /**
     * Compares the key with the pupils test, and
     * returns the number of marks scored by the pupil.
     */
    private int mark()
    {
        int score = 0;
        for (int i = 0; i < this.pupilsTest.length; i++)
        {
            if (QuizMarker.KEY[i].equals(this.pupilsTest[i]))
            {
                score = score + 1;
            }
        }
        return score;
    }
}

```



```

/**
 * Displays the result of a pupil to standard output. The argument
 * pupilScore is the number of marks the pupil scores
 */
public void reportResults(int pupilScore)
{
    int percentage = (int)((pupilScore * 100.0 / QuizMarker.KEY.length) + 0.5);
    String result = "passed";
    if (percentage < 40)
    {
        result = "failed";
    }
    System.out.println ("Percentage score is " + percentage
                        + "%" + " the pupil has " + result);
}
}

```

There are a couple of new things in this program that you might want to note. First you will see that the static variable, `KEY`, is unusually spelled in upper-case letters, and it has the keyword `final` in its header. The `final` keyword ensures that once a variable has been assigned a value or object (here it is an array object) the value or state of this variable cannot be altered during the execution of the program. To indicate this to anyone reading the program, a variable that has been declared as being static *and* final is usually spelled in upper-case letters.

The other unusual thing is that we have not provided setter or getter messages for the instance variables. This is common practice for instance variables that reference array objects – they are usually accessed directly.

Read the program through carefully, and make sure that you understand it. Note in particular the way that the `args` array in `main()` is passed as an argument to the constructor where it is assigned to the instance variable `pupilsTest`. You may also want to make sure that you understand the casting that is done in the calculation of `percentage` in the `reportResults(int)` method.

Once you understand the program, save the file as `QuizMarker.java`.

Compile your class from the command line. Now execute your program as follows:

```
java QuizMarker A B C D A B A B D C
```

Try other command line parameters.

DISCUSSION OF ACTIVITY 8

With the given test data the output is:

```
Percentage score is 20% the pupil has failed.
```

3.4 Programs that involve more than one class

In the simple examples we have used, the program has involved only one class and so the source code can be written in a single file. However, a program could involve many classes, and it may be convenient for the source code for each class to be saved in separate files. Typically, then, in real life, a Java program could consist of many files. In order to get over the inconvenience of having lots of files, the JDK allows a mechanism for collecting together the contents of all these separate files into a single zipped up file called a **.jar file**. A .jar file provides an easy and portable way of packaging programs.

Before a .jar file can be executed the programmer must indicate which class contains the `main()` method. This is recorded in a file within the .jar bundle called the **manifest file**. Once this has been done, the .jar file can be executed using the appropriate JDK command. For some operating systems the JDK will ensure that the operating system treats a .jar file as an executable file. So in Windows, for example, a .jar file can also be executed by simply double-clicking its icon.

4

Strings

You may recall that `String` objects model sequences of characters.

4.1 Creating and manipulating `String` objects

Strings are commonly encountered when a user inputs data from the keyboard. For example, you have seen assignment statements such as this:

```
name = OUDialog.request("Please enter your name", "anonymous");
```

where the expression on the right of the `=` returns a string, which is then assigned to the variable `name` (which must have been previously declared as type `String`).

You have also seen how to create a `String` object as a string literal, for example:

```
String name = "Brian";
```

Here the string referenced by `name` consists of a sequence of five characters, `'B'` `'r'` `'i'` `'a'` and `'n'`, in that order.

Underneath the surface a string is stored as an indexed sequence of characters:

0	1	2	3	4
'B'	'r'	'i'	'a'	'n'

Figure 40 A string as a sequence of characters

However, it would be a big mistake to think of strings as being similar to arrays – they are very different. Strings are constant, once a string has been created its state (the individual characters) cannot be changed. The array notation cannot be used with `String` objects. `String` objects do respond to messages, but none of those messages change a string's state.

You explored some of the protocol of `String` in *Unit 3*, where you saw useful messages such as `toUpperCase()`, `toLowerCase()` and `concat()`. In fact, `String` has a very large protocol and in the next activity you will explore some of the other messages that can be sent to instances of the `String` class.

ACTIVITY 9

Launch BlueJ and open project `Unit9_Project_1`. Then from the Tools menu select `OUWorkspace`. Open the file called `CodeForActivity9.txt`. In this activity you will look at messages in the protocol of `String` objects.

Create the following String literals.

```
String aString = "Brian Aldridge";
String bString = "Ed Grundy";
String cString = "brian aldridge";
String dString = "    a string with leading and trailing spaces    ";
```

Enter, select and execute each of the following expressions, one by one, noting the value that is returned in each case. Try to determine what effect the message has had each time. If you are not sure, try out some different examples (all of these messages will work when sent to string literals).

- 1 `aString.charAt(4);`
`bString.charAt(0);`
- 2 The following statements when executed all return an integer; do not worry about the size of the integer, just its sign (whether it is + or -).
`aString.compareTo(bString);`
`bString.compareTo(aString);`
`aString.compareTo(cString);`
`aString.compareToIgnoreCase(cString);`
- 3 `aString.indexOf('i');`
`bString.indexOf('G');`
`cString.indexOf('B');`
- 4 `aString.indexOf("Ald");`
`bString.indexOf("ward");`
- 5 `aString.replace('r', 'w');`
`bString.replace('d', 'm');`
- 6 `dString.trim();`
- 7 `aString.length();`
- 8 `aString.substring(4);`
`bString.substring(6);`
`cString.substring(3, 5);`

DISCUSSION OF ACTIVITY 9

- 1 The first message answers 'n' and the second answers 'E'. The message `charAt(int)` returns the character at the index indicated by the `int` argument.
- 2 The first message answer is a negative `int`, the second a positive `int` and the third a negative `int`. The message `compareTo(String)` compares the receiver string to the argument string. If the receiver string comes before the argument string alphabetically the message answer is a negative `int` (notice that an upper-case letter comes before the same alphabetical lower-case letter) otherwise the message answer is a positive `int`. If the two strings are exactly equal, 0 is answered. The size of the answer (positive or negative) tells you how far apart in the character sequence the first unequal characters are. So, in the first expression the answer is a -3 because 'B' in the receiver has a value that is 3 less than 'E' in the argument (the first dissimilar character in the argument). The third expression evaluates to -32 because the 'B' in the receiver has a value that is 32 less than 'b' in the argument. The fourth message expression uses the `compareToIgnoreCase(String)` message, and answers 0. As the name of the message suggests, it ignores the case of the characters and so the two strings `aString` and `cString` are deemed to be equal.

- 3 The message answer from the first expression is 2 and the message answer from the second expression is 3. The third expression returns -1. The message `indexOf(char)` returns the index within the receiver string of the first occurrence of the character argument. If the character is not found, -1 is returned.
- 4 The message answer from the first expression is 6 and the message answer from the second expression is -1. If the string argument of the message with the signature `indexOf(String)` occurs as a substring within the receiver string, then the index of the first character of the first such substring is returned; if it does not occur as a substring, -1 is returned.
- 5 The message answers in each case are "Bwian Alwidge" and "Em Grunmy". The message `replace(char, char)` returns a new string in which all the occurrences of the first `char` argument in the receiver string have been replaced by the second `char` argument. The receiver itself is not changed. If the first character does not occur in the receiver, a string with the same state as the receiver is returned.
- 6 The following string is returned "a string with leading and trailing spaces". The effect of the message `trim()` is to return a copy of the receiver string with any leading and trailing white space removed, or the receiver if it has no leading or trailing white space. In either case, the receiver is unchanged.
- 7 The message answer is 14, the number of characters in the receiver string. Note that, unlike arrays, `length()` is a message (hence the `()` is needed).
- 8 The first expression returns "n Aldridge", the second returns "ndy", and the third returns "an". The message `substring(int)` returns a new string that is a substring of the receiver string. The substring begins with the character at the index indicated by the argument and extends to the end of the receiver.

An `IndexOutOfBoundsException` is thrown if the argument is negative or greater than the length of the string. The message `substring(int int)` returns a new string that is a substring of the receiver. The substring begins at the index specified by the first `int` argument and extends to the character at an index one less than the second `int` argument (so the length of the substring is the difference between the two `int` arguments). An `IndexOutOfBoundsException` is thrown if the first argument is negative, or if it is bigger than the second argument, or if the second argument is larger than the length of the receiver string.

4.2 Equality and identity

In *Unit 3* you learnt how to use the message `equals()` to compare two strings to see if they have the same state – that is, whether they contain exactly the same characters in exactly the same order. For example, if you executed the following code:

```
String string1 = "Brian";
String string2 = "brian";
String string3 = "Brian";
```

you would expect the expression `string1.equals(string3)` to return `true` because `string1` and `string3` contain the same characters in the same order. Notice that an upper-case character and a lower-case character are considered to be different, so `string2.equals(string1)` would return `false`.

You might be surprised, however, to learn that executing `string1 == string3` also returns `true`. You will recall that the `==` operator (when used with objects) tests to see if the two operands (here `string1` and `string3`) reference the same object, so this result

You came across the identity operator in *Unit 3*.

tells us that although the code appeared to create two different `String` objects (that happened to have the same state), we, in fact, created a single `String` object, referenced by two different variables; we say that `string1` and `string3` have the same **identity**.

A reference diagram would look like this:

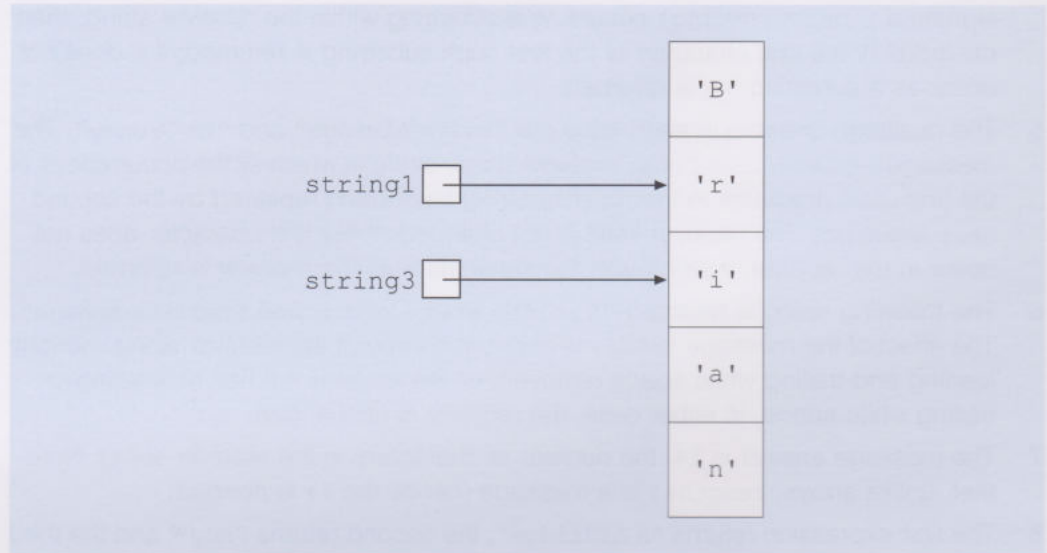


Figure 41 String variables with the same identity

In fact, whenever an attempt is made to create a string literal with exactly the same sequence of characters as an existing string literal, a second reference to the original `String` object is created instead. This means that only one copy of a particular string literal needs to be kept in the system at any one time. Each time a string literal is created Java looks to see if it already exists. If it does, rather than creating another one with exactly the same state, Java 'shares' the existing `String` object by simply creating a new reference to the existing string literal. The reason for this is one of efficiency – there is no point in having more than one string literal in the system with exactly the same character sequence, it is more efficient to share a single `String` object.

The situation is different if a string's value is computed while a program is being executed (that is, at run-time). In this case the new string becomes a distinct object, even if there is a string with the same character sequence already in the system.

Suppose the following code fragment is executed.

```
String name1 = "Jo";  
String name2 = "Lo";  
String name3 = "JoLo";  
String name4 = name1 + name2;  
String name5 = "Jo" + "Lo";  
String name6 = new String("JoLo");
```

`name4` would reference a distinct object to `name3` because `name4` references a string whose value is computed when the program is executing (we can tell this because the right-hand side of the assignment contains variables and not literal values). Similarly, `name6` and `name3` reference distinct objects, because `name6` is not assigned a literal string, but a string created by `new` at run-time.

However, `name3` and `name5` would both reference the same `String` object, "JoLo", as these strings have an identical sequence of characters, and have both been created as literals.

It is the case that the string referenced by `name4` has the same state as the string referenced by `name3`, "JoLo".

SAQ 22

Given name1, name2 etc. as created by the code above, identify three variables that reference three distinct strings with the same state.

ANSWER.....

name3, name4 and name6 all reference distinct objects with the same state (one is a literal and the other two are created at run-time). Because name5 references the same object as name3, you could have also chosen name5 instead of name3.

SAQ 23

What will Figure 41 look like after the following statement is executed?

```
String myName = "Brian";
```

ANSWER.....

An additional reference would be created to the existing String object:

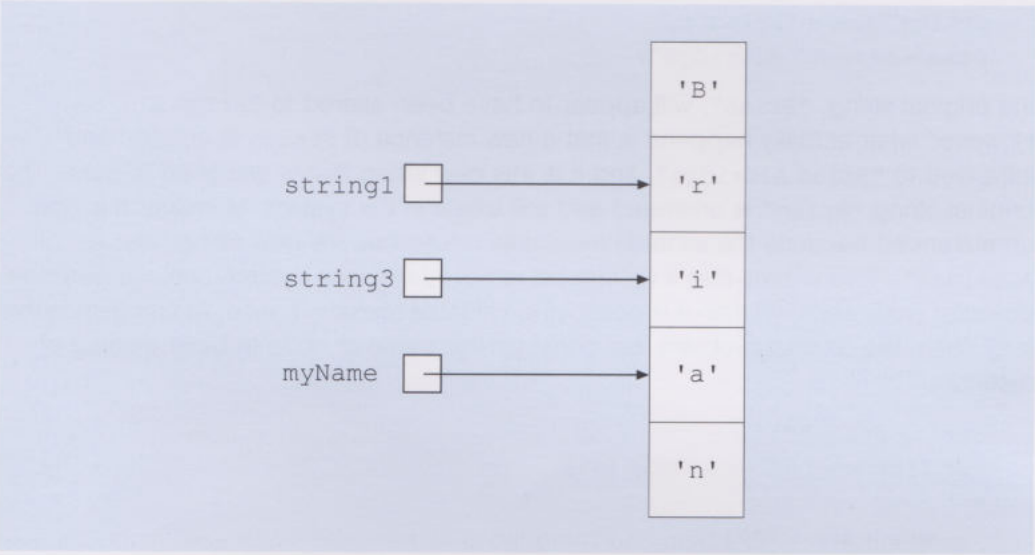


Figure 42 Answer to SAQ 23

SAQ 24

Consider the following statements.

```
String name1 = "Lee";
String name2 = "Ann";
String name3 = "LeeAnn";
String name4 = name1 + name2;
String name5 = "Lee" + "Ann";
```

Once they have all been executed, what is the result of evaluating each of the following?

- (a) `name3.equals(name4);`
- (b) `name3 == name5;`
- (c) `name3 == name4;`

ANSWER.....

- (a) true
name3 and name4 both have the same state, "LeeAnn".
- (b) true
name3 and name5 reference the same object, the single copy of the string "LeeAnn".

Important note: you may get a different result if you try this in the OUWorkspace, rather than a method because, at the time of writing, the OUWorkspace handles literal strings slightly differently from the BlueJ compiler.

(c) false

Because `name4` was created at run-time, it references a distinct object (even though a string with the same state already exists in the system).

4.3 String objects are immutable

Perhaps a surprising fact about the `String` class is that it is designed so that its instances are **immutable**, which means that once a `String` object has been created, it cannot be changed. This may seem counter-intuitive since in *Unit 3* you learnt that strings can be concatenated. For example, if this code is executed:

```
String name = "Brian";  
name = name + " Aldridge";
```

the original string, "Brian", will appear to have been altered to "Brian Aldridge". However, what actually happens is that a new instance of `String` is created and initialised to "Brian Aldridge", and it is this new string that is assigned to `name`. The original string "Brian" is unaltered and still exists in the system. Moreover it is now unreferenced because the variable `name` now references the new string "Brian Aldridge". This means that it will not be removed from the system until the garbage collector calls, and until then it is occupying valuable memory space. At first glance this may seem like a trivial problem, but consider this piece of code to build a string of asterisks:

```
String str = "";  
for (int i = 1; i <= 10000; i++)  
{  
    str = str + "*";  
}
```

Each time through the loop the previous incarnation of `str` is not deleted, instead it continues to exist as an unreferenced object. By the time that this loop has finished we have 10,001 different `String` objects ("", "*", "***", etc.) and all but one of these is unreferenced. Only the final string with 10,000 asterisks is referenced (by `str`).

In a nutshell, whenever it looks as if we are altering a string we are, in fact, creating a new string. If this new string is assigned to the original variable, the old string persists and large numbers of unreferenced objects can end up cluttering up the system (which, in turn, leads to relatively long garbage collection times). As you can imagine, unless the programmer is aware of this, the immutable property of `String` objects can lead to some very slow code. However, we will see how to get over this problem in the next section.

5 The `StringBuilder` class

In some situations the `StringBuilder` class provides a better alternative to the class `String` because a `StringBuilder` object can be changed – it is mutable. In the last section we learnt that underneath the surface a `String` object is stored as a sequence of character values. The underlying representation of a `StringBuilder` object is also a sequence of character values, however, for a `StringBuilder` object there is some **buffer** space so that the number and sequence of characters can be changed. If the buffer turns out not to be large enough to accommodate any changes, a new, longer sequence is automatically created and the characters moved over into it.

5.1 Creating `StringBuilder` objects

There are several constructors for `StringBuilder`:

- ▶ `StringBuilder()` initialises an 'empty' `StringBuilder` object with the potential to store 16 characters.
- ▶ `StringBuilder(String)` initialises an instance of `StringBuilder` that contains the same characters as the string argument plus buffer space for 16 additional characters. For example, the code:

```
StringBuilder sb = new StringBuilder("Hello");
```

declares a `StringBuilder` variable referenced by `sb`, and assigns to it a new instance of `StringBuilder` with the character sequence 'H' 'e' 'l' 'l' 'o', but with space for an additional 16 characters (giving the potential for the object referenced by `sb` to contain 21 characters in total).

A third constructor for `StringBuilder` allows the programmer to specify the **capacity** of the buffer, which is the number of characters a `StringBuilder` object can potentially hold. For example, this code creates a new `StringBuilder` object that has a capacity of 200 characters and assigns it to the variable `myStringBuilder`:

```
StringBuilder myStringBuilder = new StringBuilder(200);
```

Note that the capacity of a `StringBuilder` object is different to its length. As with a `String` object, the length of a `StringBuilder` object is the number of characters in the sequence. The capacity is the potential number of characters an existing `StringBuilder` object can hold.

Building strings with `StringBuilder`

Consider this code:

```
StringBuilder sb = new StringBuilder(10000);
for (int i = 1; i <= 10000; i++)
{
    sb.append("*");
}
```

The first line creates a new `StringBuilder` object with a buffer capacity of 10,000 and assigns it to `sb`. At this stage `sb` is 'empty'. Within the loop the `append()` message appends the character (or characters if there are more than one) contained in the string argument to the end of the existing string buffer.

In this code, only a single `StringBuilder` object ever exists. The virtual system does not have to create a new object during every iteration, as it did in the similar example we saw in Section 4.3, using a `String`. Nor does it have to garbage collect any superfluous unreferenced objects afterwards. The `StringBuilder` version of the code is very much more efficient than the `String` version.

The moral of the story is that if you are going to want to change a string, in particular if you are going to build a long string, it is far better to use a `StringBuilder` object. You may ask what happens when a `StringBuilder` buffer gets 'full'? If the buffer has reached its capacity, and an attempt is made to insert or append more characters, a new and larger underlying character sequence is created, and the existing character sequence is copied into it. The original sequence is then garbage collected. On the whole, though, letting the system create a new instance of `StringBuilder` is bad form (object creation is a time-consuming business for the processor), it is better to simply create a `StringBuilder` object that is a suitable size for your purposes in the first place. However, if you are unsure whether there is sufficient capacity in a `StringBuilder` object to append or insert additional characters, two messages can be of help. These are `capacity()`, which returns the capacity of a `StringBuilder` object; and `length()` which returns the number of characters currently in a `StringBuilder` object. The difference between these two quantities will give the amount of spare space available for additional characters.

ACTIVITY 10

We have talked about how the buffer capacity of a `StringBuilder` object makes building strings a more efficient process. How can we test that? One fairly crude but simple measure might be to compare the time it takes to build a character sequence of 10,000 asterisks using `String` and `StringBuilder` objects.

Launch BlueJ and open project `Unit9_Project_1`. Then from the Tools menu select `OUWorkspace`. Open the file called `CodeForActivity10.txt`. In this file there is the following code fragment.

```
Date d1 = new Date();
String str = "";
for (int i = 1; i <= 10000; i++)
{
    str = str + "*";
}
Date d2 = new Date();
System.out.println(d2.getTime() - d1.getTime() + " ms");
```

When executed, this code will first create an instance of `Date`, initialising the time instance variable from your computer's internal clock. Once the loop that builds a string of 10,000 asterisks has been executed, a second instance of `Date` is created. Finally, the difference in the times (in milliseconds) between these two instances of `Date` is calculated and displayed. This gives a crude reckoning of how long the loop has taken to execute.

- 1 Execute this code, and note the number of milliseconds displayed.
- 2 Write a similar code fragment that uses a `StringBuilder` object to build the character sequence of 10,000 asterisks. Note the number of milliseconds displayed when your code is executed. Note also that neither the operator `+` nor the message `concat()` can be used with `StringBuilder` objects.

DISCUSSION OF ACTIVITY 10

- 1 The number displayed will be different for different machines, and will be different from one execution to the next. (On my rather slow machine it was around 600–700 ms.)
- 2 The code looks like this:

```
Date d1 = new Date();
StringBuilder sb = new StringBuilder(10000);
for (int i = 1; i <= 10000; i++)
{
    sb.append("*");
}
Date d2 = new Date();
System.out.println(d2.getTime() - d1.getTime() + " ms");
```

(On my slow machine the time was 10 ms – much faster than the `String` version.)

Note that because the number of characters needed was known in advance, the `StringBuilder` object has been created with the correct capacity using the `StringBuilder(int)` constructor. This prevents the necessity of resizing it at run-time.

5.2 The protocol of `StringBuilder`

Many of the messages in the protocol of `StringBuilder` objects have the same signature as messages in `String`, and behave in the same way. For example, if the message with the signature `indexOf(String)` is sent to an instance of `StringBuilder` and the argument occurs as a substring within that instance of `StringBuilder`, then the index of the first character of the first such substring found is returned, just as occurs when such a message is sent to a `String` object. The messages with the signatures `substring(int)` and `substring(int, int)` also have the same effect when sent to `StringBuilder` object as to `String` object. In the following activity you will explore some of the other messages in the protocol of `StringBuilder`.

ACTIVITY 11

Launch BlueJ and open project `Unit9_Project_1`. Then from the Tools menu select `OUWorkspace`. Open the file called `CodeForActivity11.txt`.

In the `OUWorkspace` create the following `StringBuilder` object.

```
StringBuilder aSB = new StringBuilder("Bob");
```

Select and execute each of the following statements, one by one, noting the value that is returned by the message-send in each case. Try to determine what effect the message has had each time. If you are not sure try out some different examples.

- 1 `aSB.length();`
`aSB.capacity();`
- 2 `aSB.append(" is ");`
`aSB.append(50);`
- 3 `aSB.length();`
`aSB.capacity();`
- 4 `aSB.deleteCharAt(0);`
- 5 `aSB.insert(0, 'R');`
- 6 `aSB.reverse();`
- 7 `aSB.toString();`

DISCUSSION OF ACTIVITY 11

- 1 The first message returns 3 and the second 19. The default constructor creates a `StringBuilder` object that has the sequence of 3 characters that appear in the `String` argument followed by a buffer that can hold an additional 16 characters. The message `length()` returns the number of characters in the receiver `StringBuilder` object, and the message `capacity()` returns the total number of characters the `StringBuilder` object can potentially hold.
- 2 You have already seen that the message with the signature `append(String)` results in the character sequence in the `String` argument being appended to the end of the character sequence in the receiver `StringBuilder` object. The receiver is returned in its new state, containing the following character sequence `'B' 'o' 'b' ' ' 'i' 's' ' '`.

The second message, with the signature `append(int)` results in the characters `'5'` and `'0'`, corresponding to its `int` argument, being appended to the receiver `StringBuilder` object. The resulting character sequence is as follows.

```
'B' 'o' 'b' ' ' 'i' 's' ' ' '5' '0'
```

- 3 The message `length()` returns 9 (there are 9 characters, including the two blank characters). The message `capacity()` returns 19 (the receiver `StringBuilder` object can potentially hold 19 characters).
- 4 The message answer is the receiver, which has been modified by removing the character at index 0. The resulting character sequence is as follows.

```
'o' 'b' ' ' 'i' 's' ' ' '5' '0'
```

The message with the signature `deleteCharAt(int)` removes from the receiver the character at the position indicated by the message's argument and then answers with the receiver.

- 5 The message answer is the receiver, which has been modified by inserting the character `'R'` at index 0. The resulting character sequence is as follows.

```
'R' 'o' 'b' ' ' 'i' 's' ' ' '5' '0'
```

The message with the signature `insert(int, char)` results in the character given by the second argument being inserted in the receiver at the index given by the first argument. The message answers with the receiver.

- 6 The message answer is the receiver, which has been modified by reversing the character sequence. The resulting character sequence is as follows.

```
'0' '5' ' ' 's' 'i' ' ' 'b' 'o' 'R'
```

- 7 The message answer is a new `String` object `"05 si boR"` with a character sequence identical to that of the receiver `StringBuilder` object.

5.3 Revisiting the concatenation operator, +

You have learnt that the concatenation operator `+` joins two strings. Although that is true, in fact, the concatenation operator uses `StringBuilder` objects below the surface.

For example, consider this expression:

```
"a" + 5 + "b" + 10;
```

The Java compiler implements this as follows:

```
(new StringBuilder("a")).append(5).append("b").append(10).toString();
```


The fact that all the processing for the concatenations takes place in a single `StringBuilder` object, and only at the end is the result converted back into a `String`, means that far fewer temporary `String` objects have to be created, and if there are many concatenations (for example, within a loop) this internal use of `StringBuilder` means a big saving in memory and processing time.

6

Summary

After studying this unit you should understand the following ideas.

- ▶ An array is a fixed size, homogeneous indexed collection that can hold either primitive values or references to objects.
- ▶ When an array variable is declared, the type of primitive or object to be stored in the array must be stated. This is called the component type of the array.
- ▶ When an array object is created, the number of items that the array is going to store must be stated. This is called the length of the array.
- ▶ In Java, arrays are zero-based, which means that the first index is 0, so the length of an array is always one more than the last index.
- ▶ An array is a set of contiguous memory locations called components. These contain the primitive values or reference the objects that are to be stored in the array.
- ▶ A component of an array is analogous to a variable, but it is accessed using its index.
- ▶ The components of an array can be processed using looping statements.
- ▶ A two-dimensional array is an array whose components hold arrays.
- ▶ The `java.util.Arrays` class contains methods to manipulate arrays.
- ▶ A `String` object is a sequence of characters.
- ▶ A `String` object is an example of an immutable object; once a string has been created it cannot be changed.
- ▶ If many `String` objects are created using the `+` operator (within a loop, for example), problems can arise because large numbers of unreferenced strings are created that then have to be garbage collected.
- ▶ A `StringBuilder` object is also a sequence of characters, but its associated buffer space means that it can be modified. Hence `StringBuilder` objects are not immutable.

LEARNING OUTCOMES

After studying this unit you should be able to:

- ▶ describe the characteristics of an array object;
- ▶ declare and create one-dimensional and two-dimensional array objects;
- ▶ access elements in an array and use them in statements and expressions;
- ▶ process a whole array (or a meaningful sub-array) using a `foreach` statement or another looping structure, as appropriate;
- ▶ process two-dimensional array objects by row and/or by column;
- ▶ use the methods of the class `java.util.Arrays` to manipulate array objects;
- ▶ explain the difference between a `String` and a `StringBuilder` object;
- ▶ create and manipulate `String` and `StringBuilder` objects;
- ▶ explain what is meant by the term 'immutable';
- ▶ describe the problems that using `String` objects can cause, and recognise when `StringBuilder` objects should be used instead.

Glossary

.jar file A file that 'zips up' the individual files that contain the different Java classes that make up a Java program.

array An **indexable, fixed-size collection** whose **elements** are all of the same type.

buffer A sequence of unfilled **components** that allows a **StringBuilder** object to grow.

capacity The number of characters a **StringBuilder** object can hold.

collection A collection consists of a number of, possibly zero, objects or primitives. The objects or primitives within a collection are referred to as **elements**.

component The memory location at which an element (or a reference to it) of an **array** is stored.

component type The type of the **components** of an **array** object; this determines the types of the objects or primitive values that can be stored in the **array**.

effective length The number of meaningful **elements** that a **fixed-size collection** actual holds.

element The name given to the primitive value stored in, or the object referenced by, an **array component**.

final A keyword that indicates that an instance variable may not be changed once it has been assigned.

fixed-size collection A collection whose number of **elements** is fixed on creation. Such a **collection** can neither grow nor shrink.

foreach statement A statement that enables iteration through every **element** of a **collection**.

homogeneous collection A **collection** in which all the **elements** are of the same type.

identity Two variables have the same identity if they both reference the same object or primitive.

immutable An object or primitive that cannot be changed.

index An integer that is used to access the **elements** of an **indexable collection**.

indexable collection A **collection** in which every **element** is accessed by an integer **index**.

java.util.Arrays A Java utility class that contains static methods for manipulating arrays.

length The number of **elements** that an **array** can hold.

main() A public static method that all Java programs must include in order to be executed independently of an environment such as BlueJ.

String A class whose instances represent a sequence of characters. Such objects are immutable.

StringBuilder A class whose instances represent a sequence of characters. Such objects are mutable.

sub-array A part of an **array** consisting of contiguous components.

two-dimensional array An **array** whose **components** reference (one-dimensional) arrays.

Index

A

array 5, 7

B

buffer 63

C

capacity 63

collection 5, 8

component 7

component type 9, 42

E

effective length 15

element 8

element type 10, 42

F

final 55

fixed-size 7, 32

foreach statement 26

H

homogeneous 7

I

identity 60

immutable 62

index 7

indexable 7

J

.jar file 56

java.util.Arrays 46

L

length 10

M

main() 49

manifest file 56

S

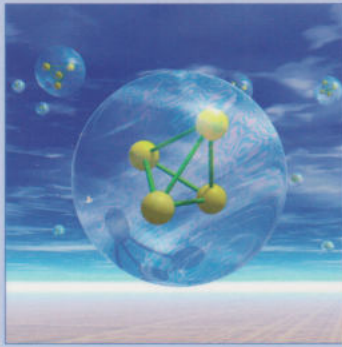
string 57

StringBuilder 63

sub-array 31

T

two-dimensional array 38



M255 Unit 9
UNDERGRADUATE COMPUTING
Object-oriented
programming with Java

UNIT

9

Block 3

- Unit 9 Collections: Arrays, strings and Stringbuilders
- Unit 10 Collections: Sets and maps
- Unit 11 Ordered and sorted collections



M255 Unit 9
ISBN 978 1 8487 3137 0

